亿级流量网站架构核心技术

跟开涛学搭建高可用高并发系统

张开涛　著

中国工信出版集团　電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.COM

亿级流量网站架构核心技术

跟开涛学搭建高可用高并发系统

张开涛　著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书总结并梳理了亿级流量网站高可用和高并发原则，通过实例详细介绍了如何落地这些原则。本书分为四部分：概述、高可用原则、高并发原则、案例实战，从负载均衡、限流、降级、隔离、超时与重试、回滚机制、压测与预案、缓存、池化、异步化、扩容、队列等多方面详细地介绍了亿级流量网站的架构核心技术，让读者看完能快速在实践中加以运用。

不管是软件开发人员还是运维人员，通过阅读本书，都能系统地学习实现亿级流量网站的关键方法与技能，并收获解决系统问题的思路和方法。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

亿级流量网站架构核心技术：跟开涛学搭建高可用高并发系统 / 张开涛著. —北京：电子工业出版社，2017.5

ISBN 978-7-121-30954-0

Ⅰ. ①亿… Ⅱ. ①张… Ⅲ. ①网站建设 Ⅳ.①TP393.092.1

中国版本图书馆 CIP 数据核字(2017)第 032588 号

审校：李希平、张凯华、刘太艳、许传奎、孙博文、王旌之、曾潇霄

策划编辑：张春雨
责任编辑：徐津平
文字编辑：杨　璐
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：30.25　字数：683 千字
版　　次：2017 年 5 月第 1 版
印　　次：2017 年 5 月第 1 次印刷
印　　数：5000 册　　定价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

书评

本书是保证大规模电商系统在高流量、高频次的冲击下仍能正常运行的葵花宝典，是互联网一线技术研发人员的实战手册。该书是经过“618”、“双 11”多次大考，在实践中反复论证应运而生的。就如山野的绿草历经大自然千锤百炼、风雨彩虹、破土而出，在自然中寻得的法则。但一切有为的成果都是辛勤努力的结果，我认识开涛后重要的印象之一就是他加班加点，挑灯夜战，几乎每天下班都是星辰相伴；印象之二是他不像传统中的 IT 男，而是一个热情、开朗、有爱心的阳光男；印象之三是他本身就如他的大作，是一个博学多才的“字典”，凡是技术性的问题大家都找他请教，有问必答。向致力于顶级电商系统建设的研发人员强烈推荐。

徐春俊 京东集团副总裁、京东保险业务负责人

经过这么多年的沉淀，京东早已摆脱“不行就加机器”的野蛮扩张阶段，今天的京东技术人有着丰富的大流量应对经验，每到大促都期望更猛烈的流量来检验他们的系统。作者集中火力讲述了他在京东构建大流量系统用到的高可用和高并发原则，并通过实际案例让读者能落地。

马松 京东集团副总裁、京东商城研发体系负责人

近十年来，京东的业务规模在不停驱动着系统的升级迭代和技术创新，到今天，京东已沉淀了不少技术创新，可以说完成了从使用技术到创新技术的转变。与此同时，京东的技术人在技术圈内的影响力也在不断扩大，开涛同学就是京东技术的一个好代表。他在网站系统升级迭代的过程中，不断创新和使用新技术，并将多年的实践和积累都浓缩到了本书中。该书可谓是当今电商互联网圈内的良心力作，理论和实践的完美结合，满满都是干货，也是京东技术人对互联网技术圈的一份贡献，强烈推荐大家阅读。

肖军 京东集团副总裁、京东 X 事业部负责人

第一次见到开涛是在部门的每周例会上，当时就对开涛留下了深刻印象，说话清晰简洁，分析严谨透彻，人也长得阳光帅气。后来才知道他在 Java 圈中知名度很高，“开涛的博客”浏览量过千万，是个不折不扣的技术大牛。本书是开涛 5 年多在高可用和高并发方面总体原则、关键技术和实战经验的总结，还包括了曾经经历的坑，可谓是理论与实践相结合的结晶。在经过了“京东 618”、“双 11”的亿级大考后，保证了此书足以作为有志于构建亿级流量网站的技术人员们必备的案头参考书。

杨建 京东保险高级研发总监

如何构建高并发、大流量的系统，不是架构师闭门造车想出来的，是线上实际的用户流量检验的。本书通过大量的实践案例，告诉读者如何架构高并发，大流量的网站系统，不光有理论探讨，亦有大量的京东实际案例，干货多，强烈推荐研发人员通读此书。

王晓钟 京东商城高级研发总监

本书内容翔实，将专业知识讲解得通俗易懂，从前端 HTML 到 DB 底层的设计无不精细阐述。更难能可贵的是，用真实成功案例传授如何在实战中进行大流量网站架构，字里行间都传递着作者的经验积累，可谓字字珠玑，是初学者的手册，更是技术大牛的切磋宝典。

尚鑫 京东商城研发总监

本书站在一个新的高度考虑网站后台技术，从应用级缓存到前端缓存、从 SOA 到闭环等无处不体现作者的深厚功底。作为京东大咖的作者结合了在京东的最佳实践，运用新的网站开发理论，提出了一套非常全面的大流量、高并发网站后台的解决方案。实践证明这一套方案特别有用，因为他结合了最新的开发技术，简化了开发过程，比较全面地考虑到了可能面临的问题。此书特别适合中大型网站的架构师、开发工程师、运维等人员，建议人手一本。

杨思勇 京东商城研发总监

首先，这是一个非常靠谱的技术人写出的非常靠谱的作品，本书作者是京东的技术牛人，长期战斗在研发的第一线，充满京东技术人的理想与激情。同时，本书也是京东这么多年高速发展经历的架构升级及大促备战经验的总结，将构建高可用、高并发系统的各种设计原则、技术方案、最佳实践进行了全面剖析，知识量非常大，值得所有大中型网站架构师、开发人员花时间学习。

王彪 京东商城研发总监

面对大流量、高并发，怎样让自己开发的系统运行得更高效、展现出更好的性能体验？系统底层怎么构建、资源怎么调度、流量怎么管控……其实这些在系统设计上都是有套路的，能将这种套路讲得特别清晰、总结得特别到位的书真的不多，此书非常值得大家一读。

付彩宝 京东商城研发总监

本书着重介绍了高并发、高可用服务的基本设计原则和技术，并辅以翔实的案例说明，对从业人员有很强的指导意义。作者开涛具备多年高并发高可用服务经验，结合自己的工作实践，将响应亿级请求的商品详情页系统的设计过程完整展现给读者，干货满满，在同类书籍中极为少见，具有很强的借鉴意义，强烈推荐。

王春明 京东商城研发总监

本书深入浅出地介绍了高并发系统的建设之路，是几年实战经验的沉淀，并且都经过了京东大促下大流量的考验。不管是初学者还是资深的架构师都能从中获取到宝贵经验。开涛是技术应用于业务、理论应用于实践的大师。开涛出品，必属精品。

何小锋 京东商城基础平台部首席架构师

大家期待已久的《亿级流量网站架构核心技术》终于出版了，这对于中国互联网界的工程师们来说真是一个天大的福利。该书可谓理论和实践结合的最佳典范，着眼于高并发和高可用，提出了一系列作者在实战中总结提炼出来的设计秘籍，并通过案例对每一条秘籍进行详细破解，书中提及的每一个案例均为作者在工作中的真实案例，都经历

过大促亿级流量的考验，全是满满的干货。该书作者开涛同学热爱技术，乐于分享，我拜读了他所有的博客和公众号文章，受益匪浅。这是作者又一次良心出品，值得研读，强烈推荐。

者文明 京东商城运营研发部首席架构师

开涛负责的京东网站等核心系统，是京东第一个迁移到京东弹性云容器平台运行的系统。在上线初期遇到架构、性能等问题，开涛以其扎实的大流量网站架构技术功底，顺利保障第一个核心系统上容器化平台。这本《亿级流量网站架构核心技术》，汇集了开涛多年在京东最核心的网站系统架构的演进和实践。特别是京东业务快速增长，对网站流量并发带来的挑战，技术选择，架构变革，最具实践意义。这本书结合实际的案例，生动展现了技术发展线路。如果你正在应对流量并发的增加或者系统架构需要变革的十字路口，这本书是你书桌上不可缺少的理论和实践指导。

鲍永成 京东商城容器引擎平台负责人

随着用户规模的增长，网站架构问题的难度也在成倍增加。构建一个京东规模的亿级流量网站和构建一个中小型网站的技术架构难度截然不同。

在具体的架构实践中，所需要考虑的问题也远比中小型网站多得多。开涛根据在京东网站架构工作期间的实战经验写成此书。书中既有大型网站架构的通用原则，也有具体难点的解决方案和实践经验。

最重要的是，书中所述的很多通用原则和技术方案都在京东网站线上得到了有效使用和验证。对于想深入了解如何构建一个大型网站的读者，这是一本难得的好书。

陈锋 京东云平台事业部架构师

读完了开涛的《亿级流量网站架构核心技术》原稿，我激动的心情难以平复，这正是我一直希望得到的那种指导手册式的技术书籍。书中没有浮夸的辞藻，而是实实在在地展示了开涛多年来在实战中验证过的理论与经验。

如果你是一位也面临着高访问高并发场景的研发人员，那么相信我，这本书中所描述的思路和方法，绝对值得你去学习和借鉴。

赵云霄 京东商城 API 网关负责人

本书详细介绍了大流量、高并发系统的设计原则和具体实现方法。从限流降级到多级缓存、异步化、服务闭环，对最近几年在高并发领域大行其道的 Nginx+Lua 架构的讲解更是细致入微。感谢开涛为大家带来这本互联网高并发架构设计的百科全书。

李尊敬 京东商城交易平台架构师

作者将多年的实践经验和研究心得呈现在这本书中，而且和实践很好地结合起来，具有很强的实践指导意义。从各个角度讲述了系统设计的注意点与优化，一层一层从前到后，范围广而详细。干劲十足，强烈推荐。

赵辉 京东商城交易平台架构师

开涛理论与实践经验结合，循序渐进地将构建亿级流量网站的高并发、高可用的一系列复杂问题阐述得很清楚。阅读此书受益匪浅，希望每一位开发人员都能阅读到这本书。

尤凤凯 京东商城交易平台架构师

本书是作者在京东商品详情页架构升级实战等多个项目中总结的成果，已经成功经历了多次“618”、“双 11”大促流量的考验，实战出真理，选择这本书，靠谱。作为技术进阶优选的书籍，满满的干货，备好水，慢慢啃。

刘峻桦 京东商城网站平台架构师

序1

开涛勤奋好学又乐于分享，他很早就深读了不少开源框架源码，吃透了内核技术，又非常喜欢看技术大侠们的分享，不断与同行交流，并学以致用，一开始参加工作就站在了较高的起点上，所以往往比同龄人做系统更加有信心，成果更加突出。他感恩于开源和分享，也践行着开源分享之路，每次埋头探索之后都有细心总结，有博客时写博客，有微信公众号时发公众号，把学到的和在实践中总结出来的，都无私分享出来。

网站是直接面对广大客户的，是公司的门户，必须快速响应，必须持续可用，必须抗得住洪峰。任何一个网站的发展过程中都出现过问题，影响客户体验和商业利益，公司业务规模越大，网站出现问题的损失越大。作者进入京东后，花了不少精力从事了“永不消失的网站”建设工作。作者和同事一起，克服了一个又一个难题，将口号变成了现实。

本书高屋建瓴，抓住了大型高并发网站设计的核心，从设计原则，到高性能、高吞吐量、高可用的系统设计，到高灵敏的监控系统构思、再到应急方案的制定，不失细节，又不拘泥于细节。相比其他已出版的关于大型网站的架构类的书籍，此书更加贴近实战，追求实用，所有内容来自于实战，文章内容也是与同道和网友们互动后改进的，本书也没有那些为了构建一个“完整的体系”而只起到填充作用的段落。此书特别适合那些快速成长型企业网站的建设者，互联网行业的研发人员，对较大规模网站的重构也有借鉴意义，看这本书可以少走些弯路，少踩些坑，其中的许多策略和技术可以直接拿来用，从而节省时间。作为本书的第一批读者，发现这本书的内容组织上兼具工具书的特点，没有严格的前后依赖，可以按章节顺序阅读，也可以随机选取中间的一章。文中公布了作者的联络方式，有问题能方便地交流。最后，希望这本书不要成为一个网站架构分享的终结者，希望有更多同学加入到探索和分享的队伍中来，不断克服新的挑战，分享更多新成果。

京东商城副总裁、京东 Y 事业部负责人 于永利

序2

我们的互联网开发者都曾经有过这样的经验。搭建一个设计精良，功能丰富的网站并不是一个高不可攀的事情。但能够支持巨大的流量而运行自如就不是一件容易的事情了。可是，当你拥有《亿级流量网站架构核心技术》这本书时，这一切又变得那么轻松。

《亿级流量网站架构核心技术》一书详细地阐述了开发高并发高可用网站的一系列关键原则问题。就如何实现系统高可用，流量高并发进行了深刻剖析。本书例举了大量的真实应用案例，帮助读者深入了解相关知识，并且使得枯燥的说教变得生动，活泼。

本书作者长期服务于京东研发的第一线，拥有丰富的软件开发经验。秉持着对技术的热爱，为互联网开发者奉献自己的心路历程。希望他的读者能够从这本书中受益。

京东集团首席技术顾问 翁志

序3

经历过“双 11”和“618”的同学都知道，在大促时如何保证系统的高并发、高可用是非常重要的事情。因此在备战大促时，有些通用原则和经验可以帮助我们在遇到高并发时，构建更可用的系统，如限流、降级、水平扩展和隔离解耦等。通过这些原则可以在流量超预期时，很好地保护系统，避免冲击导致的系统不可用。

以前京东也遇到过一些高可用问题，如超时设置不合理导致系统崩溃；限流措施不到位，导致负载过高时系统崩溃；解耦不彻底，导致某个服务挂掉时所有依赖服务受影响等。这些都是在开发和运维系统中很常见的问题，只要开发人员在开发系统时注意下这些点就可以很好地避免。书中的高可用部分可以很好地帮助读者解决这些问题。

也经常有人讨论如何提升系统性能，最直接的解决方案是扩容，或通过如加缓存来提升系统并发能力，或使用队列进行流量削峰，也可以使用异步并发机制提升吞吐量或者接口性能等。这些技术老生常谈，并不新鲜，但很实用，大家在实现高并发系统时经常会遇到。书中的高并发部分可以帮助读者理解和使用这些技术。

这本书还有一部分介绍实战案例，其中包含了京东 0 级系统“商品详情页”和“商品详情页统一服务”系统，这两个系统每天承载了京东几十亿的流量，书中深入讲解这两个系统的核心技术，还通过案例详细介绍如何使用 OpenResty 设计和开发高性能 Web 应用，值得认真阅读。

本书最大的特点是实用，书中的原则和经验是在实战中总结和进化出来的。市面上系统化地介绍高可用和高并发的文章并不多，成体系的就更少了，很多都是散落在网络上。开涛是京东优秀的架构师，有很强的架构抽象能力、扎实的编程基本功和丰富的实战经验，他将这些原则整理成体系，而且加了很多案例，相信可以很好地帮助读者学习和使用这些原则，让读者读完此书后能落地到实际项目中。

京东集团架构师 吴博

序4

大型互联网业务需要持续建设网站系统并通过 PC、移动等各种终端来与用户进行交互。大流量网站架构如何支持高并发访问并且保证高可用性，这是一个持久的、极具挑战的技术话题。毫不夸张地说，无数互联网行业的工程师为之奋斗。

开涛是京东优秀技术人才的典型代表。他从研发一线做起，脚踏实地成长为核心架构师。他所著《亿级流量网站架构核心技术》一书，分享高可用与高并发网站构建技术，干货满满，特点鲜明。

第一，理论与实践结合。本书不仅总结出一系列技术方法论，而且配合真实的案例，娓娓道来，深入浅出。读者可以直接将这些实用技术运用到自己的日常工作中。

第二，深度与广度兼具。本书选题极具针对性，专注于高可用与高并发两方面技术实践，每个方面均详解一系列技术细节。

第三，技术与业务并重。开涛并没有纯谈技术，而是围绕商品详情页——京东重要的业务产品之一，来展开更进一步的实践经验分享，给读者从业务需求到技术架构的完整视图。

第四，新兵与老将咸宜。无论是第一年人事软件开发的工程师，还是工作多年的资深人士，均可从本书中受益。

我个人强烈推荐此书。相信开涛的作品不会让大家失望。

京东商城总架构师、基础平台负责人 刘海锋

序5

去年年底我拿到本书的电子版，受邀为其写书评。全书篇幅很长，打开修订视图后，看到开涛在即将出版之前仍然在不断补充素材、示例，推敲着词句。在读到其中的某个部分的时候，我联想到当时工作中正在做的一个优化，还跟他详细讨论过，他想把这个方案也补充进去作为示例。我说，内容已经够翔实了，还嫌书不够厚吗？

是的，开涛恨不得在这本书中，一股脑儿地告诉大家他所在领域中所学到和实践的知识。写书是一个吃力还不一定能讨好的活儿，很佩服他居然能耐心写了这么多（还有很多限于整书篇幅，链接到他的博客和公众号上的扩展阅读内容）。我看到了作者的诚意。

全书前半部分我是利用坐地铁的时间看的，虽然内容我比较熟悉，但想在看书的同时如果能提前发现一些错误就更好了，看得极慢。不过，除了一些笔误之外也没发现什么硬伤。后来假期拖延症犯了，答应的书评还迟迟没有写完，后半部分就快速看过。尤其是第 4 部分案例，差不多就是开涛自己之前的工作内容，这些或多或少地都通过其他渠道看过了。

开涛结合自己的工作内容，以及相关上下游依赖系统中的各种方案、架构思想，通过自己的思考和归类总结写成本书。其中不仅有很多京东的中前端的架构实践和技术，还有作者在工作过程中用到的很多技术细节甚至代码。第 2、3 部分比较详细和系统地说明了高可用、高并发互联网应用的常用架构思想和设计方法，并配合不同的场景进行举例阐述，比较适合对此已经有了一些经验的读者。针对一些常见软件和框架的细节使用说明，以及提供很多代码的行文风格，也许能满足那些想立即动手实践的读者。

架构讲究权衡和取舍，但是前提之一是尽可能在多个相关领域的技术知识层面有经验，因此架构也很重视细节，需要对很多因素有充分思考和权衡，才有取舍。读者在阅读本书的过程中，关注点如果是各种架构方法，则需要注意作者描述的适用场景。如果关注点是各种具体的技术细节，也不要忘记思考背后所体现的架构思想。在实际的工作

中，很多方案是若干架构方法和技术的综合运用，并且随着业务或场景的变化而不断调整的，不要拘泥。

突然想到了《倚天屠龙记》中张无忌向张三丰学太极剑一节，最后张无忌成功忘记了所有的招式。对于武功高手来说，最后都是要融会贯通，形成自身体系，不要被特定的招式所束缚。学习架构，也不外乎是吧。

京东商城 交易平台架构师 肖飞

序6

动起来

开涛是个勤奋的写手，写方案，写代码，写分享，孜孜不倦。对于大多数软件开发和设计人员来说，写作不是一件容易的事。因为写出来并不是给自己看的，是要给同行们看。技术人员一方面对好的技术追求若渴，另一方面又天然地用批判和挑剔的眼光看同行的作品，算是鲁迅先生的同道吧。也正因为如此，开涛为此书内容的质量下了不少功夫。

开涛的职业生涯从空中网开始，2014 年加入京东，一头扎进了超 0 级系统的建设过程中，京东商城商品详情页改版、商品详情页统一服务规划与落地。这些系统代表着京东的形象，代表着京东技术团队的形象（开涛也是高颜值）。这些系统必须能抗峰值、不掉线、响应快，随着业务量的猛增，预期的瓶颈很快会到来，这次关键的系统改版也是从这些挑战开始，后来也就有了“永不消失的单品页”，也就有了这本书中的案例和用心总结。

作者停下开发的脚步，通过思考和总结，把动态的实践静止到了纸张上，给大家带来了精彩，愿各位读者能够把这些发生在某个历史瞬间的实践总结动态地运用到现实的开发实践中。也期望作者可以开放群或者公众号，邀请技术专家进来，与读者进行交流，动起来。

京东商城架构师 林世洪

2016 年 12 月

序7

开启探索之旅，感受技术的魅力

近年来，中国的互联网产业正在以前所未有的速度迅猛发展。而技术在业务发展中所扮演的角色日益重要，随着各个业务形态的发展涌现出了许多技术应用上的成功案例和先进技术的研究成果。而作者在本书中则通过对工作中的探索和总结来将系统高可用这个神秘莫测的面纱揭开，让对此有兴趣的人得以窥其真容。

在以往的交流和面试过程中，大多数的研发人员在其所研发的系统中很少有机会或确实不需要和繁多的上下游系统、海量的业务数据、复杂的部署环境以及极端灾难（如机房断电、光纤损坏）打交道，因此也没有契机和计划去详细了解、研究系统的高可用，对于系统高可用的理解和实践大多停留在理论认知和个人尝试阶段，很难有机会应用到解决实际业务问题上，也就很难形成自己技术和理念上的一个积累。而等到终于有机会开始在海量数据和高并发场景下一展身手的时候，又常常会因为没有系统地学习和经验积累而在设计系统、容灾策略、解决问题的过程中艰难前行。本书则通过浅显易懂的理念解读和实际案例将系统高可用相关的系统设计原则、系统限流、降级措施等“兵法三十六计”以非常直白的方式呈现给了大家。让我们对于一些常见的高并发业务场景下的系统设计原则、高可用策略有了清晰的认识和思路的拓展。无论是刚刚接触编程的学生还是已身经百战的一线研发人员都可以从书中得到很多启发，也许只是一个配置的改变、一行逻辑的优化、一个策略的调整都有可能让我们的系统可用性登上新的台阶。

京东的网站系统走过了从静态到动态、从动态到动静结合、从对 DB 的强依赖到多级缓存、从重启服务器到自如切换流量、从对 503 的恐惧到从容应对问题、从修改代码应对异常到修改配置轻松搞定的系统演变历程。当一个系统的业务体量达到可以引起系统性能和健壮性发生改变的时候，伴随着系统问题到来的更是研发人员自身能力提升和宝贵经验积累的好时机。与其将问题用重启应用和“无法解释的诡异问题”来掩盖，不如把问题的根源挖掘出来。如果挖掘得足够深入，一切问题都是可解决的。书中使用的

技术和总结的经验也许无法解决书中业务场景之外的问题，但这也恰恰是技术的魅力所在。没有一种技术和经验可以作为系统的万能解药来帮助我们一劳永逸地避免掉所有隐患，但我们可以通过对思想的接纳和消化来丰富我们的知识体系，让我们成为一个有思想的研发人员。阮一峰曾经在他的书中对于“如何变有思想”做过解释，我觉得非常适合用在研发人员的身上。研发人员的思想是什么？当你对一个需求、对一个业务形态或者对一个问题有自己的观点见解，那你就是有思想的。你的观点越多就越可能接近问题的本质，那么你的思想就越深刻和丰富。虽然你的观点不一定是事实也不一定是正确的，但作为研发人员如果有了通过不断探索、质疑、证明观点的能力之后，那么也就有了透析问题、解决问题的能力。那么在面对一个看似简单的需求或者业务时，也许你可以看得更透彻，将系统设计得更适用更合理，当你遇到书中提及的问题时也可以开始轻松应对。

我想，阅读并了解书中对于系统高可用这个领域的介绍一定会让你乐在其中。虽然你可能会有些疑惑和不解，但作为一个技术人对于技术的追求和探索不就应该是这样吗？最后，我邀请你一起踏上这个对于系统高可用的探索之旅，来感受技术的魅力。

京东商城研发总监 韩笑跃

序8

大规模分布式系统的构建，面临很多的困难和问题，但是请记住，对架构师而言，不管我们要解决多少困难，最重要的是要保证系统可用，无论任何环境、任何压力、任何场景，系统都要可用，这是我们的第一要务。在保证系统高可用的前提下，大型分布式系统面临的最突出的三大问题就是：如何应对高并发、如何处理大数据量、如何处理分布式带来的一系列问题。这也是很多一线架构老司机们的感悟和共识。

由于一本书的容量有限，不可能面面俱到，因此本书集中火力，系统、详细、专业地讲述了：大型分布式系统如何保证高可用性，以及如何应对高并发这两个大方面。涉及很多技术和细节。比如用来保证高可用的：负载均衡和反向代理、隔离、限流、降级、超时与重试等；又比如用来处理高并发的：应用缓存、多极缓存、连接池、异步并发、队列处理等。对很多朋友来说，这里面很多知识都是久闻其名，而不知其然，更不知其所以然的，学习本书正好能弥补大家在这些方面的知识短板。

作者以匠人的情怀，把每个方面从理论到应用、从技术本质到具体实现都讲得透彻明了，以平实而不失激情的风格娓娓道来，再辅以实战经验的扩展，不单单让读者学习到具体的技术和解决问题的思路，更是给出了应对问题的具体解决方案，基本上可以把这些方案拿到实际项目中直接使用。

尤为难得的是：本书还结合实际的大型应用——京东的商品详情页的实现，详细讲解了这些技术和方案在真实场景的组合应用，以更好地让知识落地。本书先是介绍了京东商品详情页的基本功能、技术架构的发展以及架构设计，当然还有很多实际的经验和体会，以“遇到的坑和问题”的面貌出现；然后详细地讲述了京东商品详情页的服务闭环实践。

为了更好地讲述京东商品详情页的具体实现，作者先讲述了实现中使用的基本技术——OpenResty，然后又详细地讲解如何使用 OpenResty 来开发商品详情页，里面涉及好多具体而细化的点，都是实际开发中会用到的，值得去认真体会。这样真实而详

细地讲述这种大型系统的实现，绝对一手的技术资料，是具有极大的参考价值的。

其实，市面上讲述大型分布式架构的书很多，但基本上都停留在理论和知识的层面，看上去都很对，很“高大上”，但就是落不了地，不能很好地跟实际应用进行结合，从而导致学习的效果欠佳。而本书很好地解决了这个问题，不仅深入浅出地讲述了各种保障高可用，以及处理高并发的技术和方案，并理论联系实际，采用京东商品详情页的具体实现这个实际案例，来综合展示了这些技术的应用，从而加深大家的理解和领悟，以更好地把这些技术和方案应用到自己的实际项目中去。

事实上，像本书这样既有详尽的技术学习，又有真实、典型案例讲述的好书，在市面上是不多见的，毕竟真正拥有这种大型系统完整架构经验的人并不多，能讲明白的更少。本书作者恰好就是那极少数技术、经验和知识传授俱佳的牛人之一，这是读者之幸。仔细阅读完本书，让人有一种醍醐灌顶的顿悟，掩卷长叹“原来如此啊”。

坦率地说，本书不是写给初学者的，对于有一定的开发经验，甚至是架构设计经验的朋友，能从本书中收获更多。但我仍然确信，不管是富有经验的架构师，还是想要学习架构知识的入门者，仔细、深入阅读本书，就一定会有收获。对于暂时不太理解的内容，建议反复阅读，或者隔段时间再看，并不断深入思考，最好是能结合实际的项目，把这些知识都应用上去，学以致用，这也不枉费作者的一番心血。

细想起来，认识作者八年多了，眼看着作者走出校园步入职场，从职场新兵，到成长成为在京东领导着上百人团队的技术大牛，仿佛一切都在昨天，让人不由不感慨时间如白驹过隙。在我眼中，作者依然是那帅气、阳光、聪明而又略微有些腼腆的大男孩形象；喜欢研究技术，特别好学、善思、勤奋，且积极在实际工作中应用所学的知识；喜欢分享技术，常年坚持撰写技术博文，拥有不少忠实粉丝，在京东内部，也是特别受欢迎的讲师之一。另外告诉大家一个小秘密，作者爱好摄影，绝对专业级水准哦。

《研磨设计模式》作者 陈臣

前言

为什么要写这本书

在 2011 年年底的时候笔者就曾规划写一本 Spring 的书，但是因为是 Spring 入门类型的书，框架的内容更新太快，觉得还是写博客好一些，因此就把写完的书稿《跟我学 Spring》放到了博客（jinnianshilongnian.iteye.com，因为是龙年开的博客，很多网友喊我龙年兄）中，并持续更新，到现在已经差不多五年了。大家在网上找资源时会发现，很多内容不成体系，不能用来系统地学习，这也是我曾经的痛点，因此我写博客的一个特色就是坚持写系列文章——想学习某种技术只要我的博客有就不需要去其他地方再找了，到现在已经写过《跟我学 Spring》、《Spring 杂谈》、《跟我学 Spring MVC》、《跟我学 Shiro》、《跟我学 Nginx+Lua》等系列，累计访问量已超过 1000 万。我写博客还有一个私心：带新人，当时我们系统架构使用 OpenResty，而团队成员都是 Java 程序员，所以就写了《跟我学 OpenResty（Nginx+Lua）开发》，新人跟着教程学一遍就能上手干活了。扫一扫关注我的博客。

2015 年开始，笔者在个人公众号“开涛的博客”撰写《聊聊高并发系统》系列文章，陆续发表了《聊聊高并发系统之限流特技》、《聊聊高并发系统之降级特技》、《聊聊高并发系统之队列术》、《构建需求响应式亿级商品详情页》等文章。这些内容都是笔者在一线使用过的一些技能，而这些技能又是一线程序员或架构师应该掌握的必备技能。而且

这一系列也得到了很多读者的反馈和认可，帮助他们解决了系统的一些问题。公众号发表的有些内容偏理论，很多人不知道怎么去用，因此就有了丰富理论和实战内容并出版本书的想法。想学习高可用和高并发系统技能，看这本书就够了，并且可以作为案头工具书来用。

笔者耗费了大半年业余时间才成就此书，希望这些实战中能真地用得上的技术可以帮助到读者。

本书讲解的原则并不是笔者总结出来的，有许许多多前辈们已经实践过，笔者只是花了点时间进行汇总，并把工作中使用过的一些经验和案例融入到书中。

成长和进步是一个循序渐进的过程，妄图看完本书后能屠龙降魔是不可能的，别人走过的路还是会走一遍，别人踩过的坑还是会踩一遍。正如作家格拉德威尔在《异类：不一样的成功启示录》一书中的一万小时定律：“人们眼中的天才之所以卓越非凡，并非天资超人一等，而是付出了持续不断的努力。一万小时的锤炼是任何人从平凡变成世界级大师的必要条件”。

读者对象

本书希望对在一线从事开发工作或正在解决一线问题的朋友有所帮助。

如何阅读本书

本书的内容是理论与实战相结合，涉及的知识点比较多，共分为4个部分，读者可按照任何顺序阅读每一个部分，但建议先阅读第1部分进行系统了解。

第1部分概述，主要介绍开发高并发系统的一些原则，并阐述本书将要讲解的原则。

第2部分高可用，帮助读者理解高可用的一些原则，如负载均衡、限流、降级、隔离、超时与重试、回滚机制、压测与预案等，并能实际应用到自己的系统中。

第3部分高并发，介绍开发高并发系统的一些原则，如缓存、池化、异步化、扩容、队列等，并配合大量案例帮助读者更好地掌握和运用。

第4部分案例，介绍笔者开发过的商品详情页、统一服务等系统架构，还有一些静态化架构的思路，帮助读者理解前边介绍的一些原则。

阅读本书需要对Java、OpenResty（Nginx+Lua）、Redis、MysQl等技术有一定了解，OpenResty可以参考我的博客中的《跟我学OpenResty（Nginx+Lua）开发》系列文章。本

文提到的 Nginx+Lua 等同于 OpenResty。可扫码阅读《跟我学 OpenResty（Nginx+Lua）开发》。

因为篇幅原因，本书示例很难做到全面且详细，因此思路不要受限于书中所写，要活学活用，举一反三。比如多级缓存的思路，可以扩展到多级存储：内存→NVMe/SATA SSD→机械盘。

勘误和支持

由于笔者能力有限，虽然找了很多朋友帮忙校对，但书中难免会出现一些错误，也请读者朋友批评指正。大家可以扫如下二维码关注我的公众号或者访问我的博客留言反馈错误和建议，笔者会积极提供解答。

致谢

首先要感谢进入京东商城时的架构组的同事们，感谢隋剑峰、邹开红、冯培源、李尊敬、徐涛、杨超、王战兵、赵辉、孙炳蔚等对我的帮助，也感谢杨思勇、尚鑫、徐烁、韩笑跃等对我的信任，并给了我大胆实践商城单品页的机会，还有我的好搭档刘峻桦，还有王晓钟、刘海峰、林世洪、肖飞、何小锋、鲍永成、刘行、周昱行等对我的帮助和

支持，感谢我的领导徐春俊、杨建对我的支持和肯定，感谢京东和我的团队，还有许许多多一起合作过和交流过的朋友们，没有你们的帮助就没有这本书的出版。

感谢张志统、肖飞、赵云霄、马顺风、刘兵、张亮、颜晟、曾波、孙伟、王景、黄杨俊、王君富、李晋、刘嘉南、刘艺飞、吴正轩、邵东风、孙鹏、张金立、任敬表、刘冉、陈玉苗、王晓雯、李乐伟、晁志刚、王向维、赵湘建、尤凤凯等对本书的校对和建议。感谢林世洪、肖飞、赵云霄为本书提供素材。也感谢那些在我博客和公众号留言和鼓励我的朋友，最后感谢电子工业出版社的侠少和杨璐的支持。

目录

第 1 部分　概述 1

1　交易型系统设计的一些原则 2

1.1　高并发原则 3

1.1.1　无状态 3

1.1.2　拆分 3

1.1.3　服务化 4

1.1.4　消息队列 4

1.1.5　数据异构 6

1.1.6　缓存银弹 7

1.1.7　并发化 9

1.2　高可用原则 10

1.2.1　降级 10

1.2.2　限流 11

1.2.3　切流量 12

1.2.4　可回滚 12

1.3　业务设计原则 12

1.3.1　防重设计 13

1.3.2　幂等设计 13

1.3.3　流程可定义 13

1.3.4　状态与状态机 13

1.3.5　后台系统操作可反馈 14

1.3.6　后台系统审批化 14

1.3.7　文档和注释 14

1.3.8 备份 14
1.4 总结 14

第 2 部分 高可用 17

2 负载均衡与反向代理 18

2.1 upstream 配置 20
2.2 负载均衡算法 21
2.3 失败重试 23
2.4 健康检查 24
2.4.1 TCP 心跳检查 24
2.4.2 HTTP 心跳检查 24
2.5 其他配置 25
2.5.1 域名上游服务器 25
2.5.2 备份上游服务器 26
2.5.3 不可用上游服务器 26
2.6 长连接 26
2.7 HTTP 反向代理示例 29
2.8 HTTP 动态负载均衡 30
2.8.1 Consul+Consul-template 31
2.8.2 Consul+OpenResty 35
2.9 Nginx 四层负载均衡 39
2.9.1 静态负载均衡 39
2.9.2 动态负载均衡 41
参考资料 42

3 隔离术 43

3.1 线程隔离 43
3.2 进程隔离 45
3.3 集群隔离 45
3.4 机房隔离 46
3.5 读写隔离 47

3.6 动静隔离 48
3.7 爬虫隔离 49
3.8 热点隔离 50
3.9 资源隔离 50
3.10 使用 Hystrix 实现隔离 51
3.10.1 Hystrix 简介 51
3.10.2 隔离示例 52
3.11 基于 Servlet 3 实现请求隔离 56
3.11.1 请求解析和业务处理线程池分离 57
3.11.2 业务线程池隔离 58
3.11.3 业务线程池监控/运维/降级 58
3.11.4 如何使用 Servlet 3 异步化 59
3.11.5 一些 Servlet 3 异步化压测数据 64

4 限流详解 66
4.1 限流算法 67
4.1.1 令牌桶算法 67
4.1.2 漏桶算法 68
4.2 应用级限流 69
4.2.1 限流总并发/连接/请求数 69
4.2.2 限流总资源数 70
4.2.3 限流某个接口的总并发/请求数 70
4.2.4 限流某个接口的时间窗请求数 70
4.2.5 平滑限流某个接口的请求数 71
4.3 分布式限流 75
4.3.1 Redis+Lua 实现 76
4.3.2 Nginx+Lua 实现 77
4.4 接入层限流 78
4.4.1 ngx_http_limit_conn_module 78
4.4.2 ngx_http_limit_req_module 80
4.4.3 lua-resty-limit-traffic 88
4.5 节流 90

4.5.1 throttleFirst/throttleLast 90
4.5.2 throttleWithTimeout 91
参考资料 92

5 降级特技 93

5.1 降级预案 93
5.2 自动开关降级 95
5.2.1 超时降级 95
5.2.2 统计失败次数降级 95
5.2.3 故障降级 95
5.2.4 限流降级 95
5.3 人工开关降级 96
5.4 读服务降级 96
5.5 写服务降级 97
5.6 多级降级 98
5.7 配置中心 100
5.7.1 应用层 API 封装 100
5.7.2 使用配置文件实现开关配置 101
5.7.3 使用配置中心实现开关配置 102
5.8 使用 Hystrix 实现降级 106
5.9 使用 Hystrix 实现熔断 108
5.9.1 熔断机制实现 108
5.9.2 配置示例 112
5.9.3 采样统计 113

6 超时与重试机制 117

6.1 简介 117
6.2 代理层超时与重试 119
6.2.1 Nginx 119
6.2.2 Twemproxy 126
6.3 Web 容器超时 127
6.4 中间件客户端超时与重试 128

6.5 数据库客户端超时 ... 131
6.6 NoSQL 客户端超时 ... 134
6.7 业务超时 ... 135
6.8 前端 Ajax 超时 ... 135
6.9 总结 ... 136
6.10 参考资料 ... 137

7 回滚机制 ... 139

7.1 事务回滚 ... 139
7.2 代码库回滚 ... 140
7.3 部署版本回滚 ... 141
7.4 数据版本回滚 ... 142
7.5 静态资源版本回滚 ... 143

8 压测与预案 ... 145

8.1 系统压测 ... 145
8.1.1 线下压测 ... 146
8.1.2 线上压测 ... 146
8.2 系统优化和容灾 ... 147
8.3 应急预案 ... 148

第 3 部分 高并发 ... 153

9 应用级缓存 ... 154

9.1 缓存简介 ... 154
9.2 缓存命中率 ... 155
9.3 缓存回收策略 ... 155
9.4 Java 缓存类型 ... 156
9.4.1 堆缓存 ... 158
9.4.2 堆外缓存 ... 161
9.4.3 磁盘缓存 ... 162
9.4.4 分布式缓存 ... 163

9.4.5 多级缓存 166
9.5 应用级缓存示例 167
9.5.1 多级缓存 API 封装 167
9.5.2 NULL Cache 170
9.5.3 强制获取最新数据 170
9.5.4 失败统计 171
9.5.5 延迟报警 171
9.6 缓存使用模式实践 172
9.6.1 Cache-Aside 173
9.6.2 Cache-As-SoR 174
9.6.3 Read-Through 174
9.6.4 Write-Through 176
9.6.5 Write-Behind 177
9.6.6 Copy Pattern 181
9.7 性能测试 181
9.8 参考资料 182

10 HTTP 缓存 183

10.1 简介 183
10.2 HTTP 缓存 184
10.2.1 Last-Modified 184
10.2.2 ETag 190
10.2.3 总结 192
10.3 HttpClient 客户端缓存 192
10.3.1 主流程 195
10.3.2 清除无效缓存 195
10.3.3 查找缓存 196
10.3.4 缓存未命中 198
10.3.5 缓存命中 198
10.3.6 缓存内容陈旧需重新验证 202
10.3.7 缓存内容无效需重新执行请求 205
10.3.8 缓存响应 206
10.3.9 缓存头总结 207

10.4 Nginx HTTP 缓存设置 208
10.4.1 expires 208
10.4.2 if-modified-since 209
10.4.3 nginx proxy_pass 209
10.5 Nginx 代理层缓存 211
10.5.1 Nginx 代理层缓存配置 211
10.5.2 清理缓存 214
10.6 一些经验 215
参考资料 216

11 多级缓存 217

11.1 多级缓存介绍 217
11.2 如何缓存数据 219
11.2.1 过期与不过期 219
11.2.2 维度化缓存与增量缓存 220
11.2.3 大 Value 缓存 220
11.2.4 热点缓存 220
11.3 分布式缓存与应用负载均衡 221
11.3.1 缓存分布式 221
11.3.2 应用负载均衡 221
11.4 热点数据与更新缓存 222
11.4.1 单机全量缓存+主从 222
11.4.2 分布式缓存+应用本地热点 223
11.5 更新缓存与原子性 224
11.6 缓存崩溃与快速修复 225
11.6.1 取模 225
11.6.2 一致性哈希 225
11.6.3 快速恢复 225

12 连接池线程池详解 227

12.1 数据库连接池 227
12.1.1 DBCP 连接池配置 228

12.1.2 DBCP 配置建议 233
12.1.3 数据库驱动超时实现 234
12.1.4 连接池使用的一些建议 235
12.2 HttpClient 连接池 236
12.2.1 HttpClient 4.5.2 配置 236
12.2.2 HttpClient 连接池源码分析 240
12.2.3 HttpClient 4.2.3 配置 241
12.2.4 问题示例 243
12.3 线程池 244
12.3.1 Java 线程池 245
12.3.2 Tomcat 线程池配置 248

13 异步并发实战 250
13.1 同步阻塞调用 251
13.2 异步 Future 252
13.3 异步 Callback 253
13.4 异步编排 CompletableFuture 254
13.5 异步 Web 服务实现 257
13.6 请求缓存 259
13.7 请求合并 261

14 如何扩容 266
14.1 单体应用垂直扩容 267
14.2 单体应用水平扩容 267
14.3 应用拆分 268
14.4 数据库拆分 271
14.5 数据库分库分表示例 275
14.5.1 应用层还是中间件层 275
14.5.2 分库分表策略 277
14.5.3 使用 sharding-jdbc 分库分表 279
14.5.4 sharding-jdbc 分库分表配置 279
14.5.5 使用 sharding-jdbc 读写分离 283

14.6 数据异构 284
14.6.1 查询维度异构 284
14.6.2 聚合据异构 285
14.7 任务系统扩容 285
14.7.1 简单任务 285
14.7.2 分布式任务 287
14.7.3 Elastic-Job 简介 287
14.7.4 Elastic-Job-Lite 功能与架构 287
14.7.5 Elastic-Job-Lite 示例 288

15 队列术 295

15.1 应用场景 295
15.2 缓冲队列 296
15.3 任务队列 297
15.4 消息队列 297
15.5 请求队列 299
15.6 数据总线队列 300
15.7 混合队列 301
15.8 其他队列 302
15.9 Disruptor+Redis 队列 303
15.9.1 简介 303
15.9.2 XML 配置 304
15.9.3 EventWorker 305
15.9.4 EventPublishThread 307
15.9.5 EventHandler 308
15.9.6 EventQueue 308
15.10 下单系统水平可扩展架构 311
15.10.1 下单服务 312
15.10.2 同步 Worker 313
15.11 基于 Canal 实现数据异构 314
15.11.1 MySQL 主从复制 315
15.11.2 Canal 简介 316

15.11.3 Canal 示例 318

第 4 部分 案例 323

16 构建需求响应式亿级商品详情页 324

16.1 商品详情页是什么 324
16.2 商品详情页前端结构 325
16.3 我们的性能数据 327
16.4 单品页流量特点 327
16.5 单品页技术架构发展 327
16.5.1 架构 1.0 328
16.5.2 架构 2.0 328
16.5.3 架构 3.0 330
16.6 详情页架构设计原则 332
16.6.1 数据闭环 332
16.6.2 数据维度化 333
16.6.3 拆分系统 334
16.6.4 Worker 无状态化+任务化 334
16.6.5 异步化+并发化 335
16.6.6 多级缓存化 335
16.6.7 动态化 336
16.6.8 弹性化 336
16.6.9 降级开关 336
16.6.10 多机房多活 337
16.6.11 两种压测方案 338
16.7 遇到的一些坑和问题 339
16.7.1 SSD 性能差 339
16.7.2 键值存储选型压测 339
16.7.3 数据量大时 JIMDB 同步不动 342
16.7.4 切换主从 342
16.7.5 分片配置 342
16.7.6 模板元数据存储 HTML 342
16.7.7 库存接口访问量 600w/分钟 343

16.7.8 微信接口调用量暴增 344
16.7.9 开启 Nginx Proxy Cache 性能不升反降 344
16.7.10 配送至读服务因依赖太多，响应时间偏慢 344
16.7.11 网络抖动时，返回 502 错误 346
16.7.12 机器流量太大 346
16.8 其他 347

17 京东商品详情页服务闭环实践 348

17.1 为什么需要统一服务 348
17.2 整体架构 349
17.3 一些架构思路和总结 350
17.3.1 两种读服务架构模式 351
17.3.2 本地缓存 352
17.3.3 多级缓存 352
17.3.4 统一入口/服务闭环 353
17.4 引入 Nginx 接入层 354
17.4.1 数据校验/过滤逻辑前置 354
17.4.2 缓存前置 354
17.4.3 业务逻辑前置 355
17.4.4 降级开关前置 355
17.4.5 A/B 测试 355
17.4.6 灰度发布/流量切换 356
17.4.7 监控服务质量 356
17.4.8 限流 356
17.5 前端业务逻辑后置 356
17.6 前端接口服务器端聚合 357
17.7 服务隔离 358

18 使用 OpenResty 开发高性能 Web 应用 360

18.1 OpenResty 简介 361
18.1.1 Nginx 优点 361
18.1.2 Lua 的优点 361

18.1.3 什么是 ngx_lua 361
18.1.4 开发环境 362
18.1.5 OpenResty 生态 362
18.1.6 场景 362
18.2 基于 OpenResty 的常用架构模式 363
18.2.1 负载均衡 363
18.2.2 单机闭环 364
18.2.3 分布式闭环 367
18.2.4 接入网关 368
18.2.5 Web 应用 370
18.3 如何使用 OpenResty 开发 Web 应用 371
18.3.1 项目搭建 371
18.3.2 启停脚本 372
18.3.3 配置文件 372
18.3.4 Nginx.conf 配置文件 372
18.3.5 Nginx 项目配置文件 373
18.3.6 业务代码 374
18.3.7 模板 374
18.3.8 公共 Lua 库 374
18.3.9 功能开发 375
18.4 基于 OpenResty 的常用功能总结 375
18.5 一些问题 376

19 应用数据静态化架构高性能单页 Web 应用 377

19.1 整体架构 378
19.1.1 CMS 系统 379
19.1.2 前端展示系统 379
19.1.3 控制系统 380
19.2 数据和模板动态化 381
19.3 多版本机制 381
19.4 异常问题 381

20 使用 OpenResty 开发 Web 服务 383

20.1 架构 383
20.2 单 DB 架构 384
20.2.1 DB+Cache/数据库读写分离架构 384
20.2.2 OpenResty+Local Redis+MySQL 集群架构 385
20.2.3 OpenResty+Redis 集群+MySQL 集群架构 386
20.3 实现 387
20.3.1 后台逻辑 388
20.3.2 前台逻辑 388
20.3.3 项目搭建 389
20.3.4 Redis+Twemproxy 配置 389
20.3.5 MySQL+Atlas 配置 390
20.3.6 Java+Tomcat 安装 394
20.3.7 Java+Tomcat 逻辑开发 395
20.3.8 Nginx+Lua 逻辑开发 401

21 使用 OpenResty 开发商品详情页 405

21.1 技术选型 407
21.2 核心流程 408
21.3 项目搭建 408
21.4 数据存储实现 410
21.4.1 商品基本信息 SSDB 集群配置 410
21.4.2 商品介绍 SSDB 集群配置 413
21.4.3 其他信息 Redis 配置 416
21.4.4 集群测试 418
21.4.5 Twemproxy 配置 419
21.5 动态服务实现 421
21.5.1 项目搭建 422
21.5.2 项目依赖 422
21.5.3 核心代码 422
21.5.4 web.xml 配置 427
21.5.5 打 WAR 包 428

21.5.6 配置 Tomcat 428
21.5.7 测试 428
21.5.8 Nginx 配置 428
21.5.9 绑定 hosts 测试 430
21.6 前端展示实现 430
21.6.1 基础组件 430
21.6.2 商品介绍 432
21.6.3 前端展示 433
21.6.4 测试 441
21.6.5 优化 441

第 1 部分　概述

- 高并发原则
- 高可用原则
- 业务设计原则
- 总结

1 交易型系统设计的一些原则

在我们的技术生涯中，总是不断针对新的需求去研发新的系统，而很多系统的设计都是可以触类旁通的。在设计系统时，要因场景、时间而异，一个系统也不是一下子就能设计得非常完美，在具有有限资源的情况下，一定是先解决当下最核心的问题，预测并发现未来可能出现的问题，一步步解决最痛点的问题。也就是说，系统设计是一个不断迭代的过程，在迭代中发现问题并修复问题，即满足需求的系统是不断迭代优化出来的，这是一个持续的过程，个人不相信完美架构银弹。不过，如果一开始就有好的基础系统设计，未来可以更容易达到一个比较满意的目标。一个好的设计要做到，解决现有需求和问题，把控实现和进度风险，预测和规划未来，不要过度设计，从迭代中演进和完善。

在设计系统时，应该多思考**墨菲定律**。

1. 任何事都没有表面看起来那么简单。
2. 所有的事都会比你预计的时间长。
3. 可能出错的事总会出错。
4. 如果你担心某种情况发生，那么它就更有可能发生。

在系统划分时，也要思考**康威定律**。

1. 系统架构是公司组织架构的反映。

2. 应该按照业务闭环进行系统拆分/组织架构划分，实现闭环/高内聚/低耦合，减少沟通成本。

3. 如果沟通出现问题，那么就应该考虑进行系统和组织架构的调整。

4. 在合适时机进行系统拆分，不要一开始就把系统/服务拆得非常细，虽然闭环，但是每个人维护的系统多，维护成本高。

应该多鼓励团队成员积极主动沟通并推动系统演进。另外，也要多思考**二八定律**，在系统设计初期将有限的资源用到刀刃上，以最小化可行产品方式迭代推进。

在持续开发系统的过程中，会有一些设计原则/经验可以用来遵循和指导我们。但设计原则应该在系统迭代过程中，根据现有问题或特征匹配使用，如果刚开始遇到的不是核心问题，那么不要复杂化系统设计，但先行规划和设计是有必要的，要对现有问题有方案，对未来架构有预案。

1.1 高并发原则

1.1.1 无状态

如果设计的应用是无状态的，那么应用比较容易进行水平扩展。实际生产环境可能是这样的：应用无状态，配置文件有状态。比如，不同的机房需要读取不同的数据源，此时，就需要通过配置文件或配置中心指定。

1.1.2 拆分

在系统设计初期，是做一个大而全的系统还是按功能模块拆分系统，这个需要根据环境进行权衡。比如，做私塾在线时，本身用户量/交易量不会特别大，开发就笔者一个人，资源有限，那就没必要对系统拆分（比如，拆分商品、订单等），做一个大而全的系统即可。而像设计京东秒杀系统，访问量是非常大的，而且投入的资源还是蛮充足的，在这种情况下，就可以考虑按功能拆分系统。

笔者遇到的拆分主要有如下几种情况。

系统维度：按照系统功能/业务拆分，比如商品系统、购物车、结算、订单系统等。

功能维度：对一个系统进行功能再拆分，比如，优惠券系统可以拆分为后台券创建系统、领券系统、用券系统等；

读写维度：根据读写比例特征进行拆分。比如，商品系统，交易的各个系统都会读取数据，读的量大于写，因此可以拆分成商品写服务、商品读服务；读服务可以考虑使用缓存提升性能；写的量太大时，需要考虑分库分表；有些聚合读取的场景，如商品详情页，可考虑数据异构拆分系统，将分散在多处的数据聚合到一处存储，以提升系统的性能和可靠性；

AOP 维度：根据访问特征，按照 AOP 进行拆分，比如，商品详情页可以分为 CDN、页面渲染系统；CDN 就是一个 AOP 系统。

模块维度：比如，按照基础或者代码维护特征进行拆分，如基础模块分库分表、数据库连接池等；代码结构一般按照三层架构（Web、Service、DAO）进行划分。

1.1.3 服务化

首先，判断是不是只需要简单的单点远程服务调用，单机不行集群是不是就可以解决？在客户端注册多台机器并使用Nginx进行负载均衡是不是就可以解决？随着调用方越来越多，应该考虑使用服务自动注册和发现（如 Dubbo 使用 ZooKeeper）。其次，还要考虑服务的分组/隔离，比如，有的系统访问量太大，导致把整个服务打挂，因此，需要为不同的调用方提供不同的服务分组，隔离访问。后期随着调用量的增加还要考虑服务的限流、黑白名单等。还有一些细节需要注意，如超时时间、重试机制、服务路由（能动态切换不同的分组）、故障补偿等，这些都会影响到服务的质量。

总结为：进程内服务→单机远程服务→集群手动注册服务→自动注册和发现服务→服务的分组/隔离/路由→服务治理如限流/黑白名单。

1.1.4 消息队列

消息队列是用来解耦一些不需要同步调用的服务或者订阅一些自己系统关心的变化。使用消息队列可以实现服务解耦（一对多消费）、异步处理、流量削峰/缓冲等。比如，电商系统中的交易订单数据，该数据有非常多的系统关心并订阅，比如，订单生产系统、定期送系统、订单风控系统等等。如果订阅者太多，那么订阅单个消息队列就会

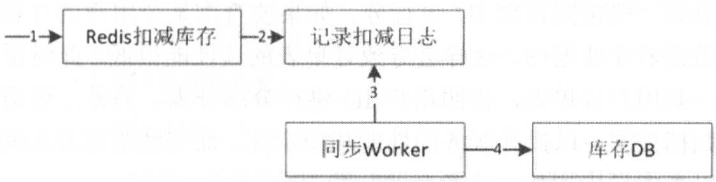
成为瓶颈，此时，需要考虑对消息队列进行多个镜像复制。

使用消息队列时，还要注意处理生产消息失败，以及消息重复接收时的场景。有些消息队列产品会提供生产重试功能，在达到指定重试次数还未生产成功时，会对外通知生产失败。这时，对于不能容忍生产失败的业务场景来说，一定要做好后续的数据处理工作，如持久化数据要同时增加日志、报警等。对于消息重复问题，特别是一些分布式消息队列，出于对性能和开销的考虑，在一些场景下会发生消息重复接收，需要在业务层面进行防重处理。

1. 大流量缓冲

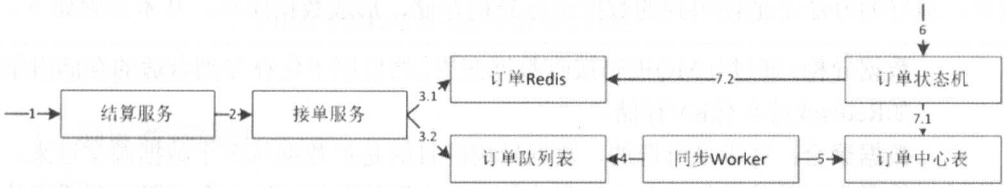
在电商搞大促时，系统流量会高于正常流量的几倍甚至几十倍，此时就要进行一些特殊的设计来保证系统平稳度过这段时期；而解决的手段很多，一般都是牺牲强一致性，而保证最终一致性即可。

比如，扣减库存，可以考虑这样设计。



直接在 Redis 中扣减，然后记录下扣减日志，通过 Worker 同步到 DB。

还有，如交易订单系统，可以考虑这样设计。



首先，结算服务调用订单接单服务，将订单存储到订单 Redis 和订单队列表，订单队列表可以按照需求水平扩展多个表，通过队列缓冲表提升接单能力。然后，通过同步 Worker 同步到订单中心表；假设用户支付了订单，订单状态机会驱动状态变更，此时，可能订单队列表的订单还没有同步到订单中心表，状态机要根据实际情况进行重试。

如果用户查看单个订单详情，那么可以直接从订单 Redis 中查到。但如果查询订单

列表，则需要考虑订单 Redis 和列表的合并。

同步 Worker 在设计时，需要考虑并发处理和重复处理的问题，比如，使用单机串行扫描处理（每台 Worker 只扫描其中的一部分表）还是集群处理（Map-Reduce）。另外，需要考虑是否需要对订单队列表添加相关字段：处理人（哪个应用正在处理）和处理状态（正在处理、已处理、处理失败）、最后处理时间（应对超时）、失败次数等。

2. 数据校对

在使用了消息异步机制的场景下，可能存在消息的丢失，需要考虑进行数据校对和修正来保证数据的一致性和完整性。可以通过 Worker 定期去扫描原始表，通过对业务数据进行校对，有问题的要进行补偿，扫描周期根据实际场景进行定义。

1.1.5 数据异构

1. 数据异构

订单分库分表一般按照订单 ID 进行分，如果要查询某个用户的订单列表，则需要聚合多个表的数据后才能返回，这样会导致订单表的读性能很低。此时需要对订单表进行异构，异构一套用户订单表，按照用户 ID 进行分库分表。另外，还需要考虑对历史订单数据进行归档处理，以提升服务的性能和稳定性。而有些数据异构的意义不大，如库存价格，可以考虑异步加载，或者合并并发请求。

2. 数据闭环

数据闭环如商品详情页，因为数据来源太多，影响服务稳定性的因素就非常多了。因此，最好的办法是把使用到的数据进行异构存储，形成数据闭环，基本步骤如下。

- **数据异构：** 通过如MQ机制接收数据变更，然后原子化存储到合适的存储引擎，如Redis或持久化KV存储。
- **数据聚合：** 这步是可选的，数据异构的目的是把数据从多个数据源拿过来，数据聚合的目的是把这些数据做个聚合，这样前端就可以一个调用拿到所有数据，此步骤一般存储到KV存储中。
- **前端展示：** 前端通过一次或少量几次调用拿到所需要的数据。

这种方式的好处就是数据的闭环，任何依赖系统出问题了，还是能正常工作，只是更新会有积压，但是不影响前端展示。

另外，此处如果一次需要多个数据，那么可以考虑使用 Hash Tag 机制将相关的数据

聚合到一个实例，如在展示商品详情页时需要商品基本信息“p:productId:”和商品规格参数“d:productId:”，此时就可以使用冒号中间的 productId 作为数据分片 key，这样相同 productId 的商品相关数据就在一个实例。

数据闭环和数据异构其实是一个概念，目的都是实现数据的自我控制，当其他系统出问题时不影响自己的系统，或者自己出问题时不影响其他系统。一般通过消息队列来实现数据分发。

1.1.6 缓存银弹

缓存对于读服务来说可谓抗流量的银弹，可总结为下表。

流程节点	缓存技术
客户端	使用浏览器缓存
	客户端应用缓存
客户端网络	代理服务器开启缓存
广域网	使用代理服务器(含 CDN）
	使用镜像服务器
	使用 P2P 技术
源站及源站网络	使用接入层提供的缓存机制
	使用应用层提供的缓存机制
	使用分布式缓存
	静态化、伪静态化
	使用服务器操作系统提供的缓存机制

本表由林世洪提供。

1. 浏览器端缓存

设置请求的过期时间，如对响应头 Expires、Cache-control 进行控制。这种机制适用于对实时性不太敏感的数据，如商品详情页框架、商家评分、评价、广告词等；但对于价格、库存等实时要求比较高的数据，就不能做浏览器端缓存。

2. APP 客户端缓存

在大促时为了防止瞬间流量冲击，一般会在大促之前把 APP 需要访问的一些素材

（如 js/css/image 等）提前下发到客户端进行缓存，这样在大促时就不用去拉取这些素材了。还有如首屏数据也可以缓存起来，在网络异常情况下还是有托底数据给用户展示；还有如 APP 地图一般也会做地图的离线缓存。

3. CDN 缓存

有些页面、活动页、图片等服务可以考虑将页面、活动页、图片推送到离用户最近的 CDN 节点，让用户能在离他最近的节点找到想要的数据。一般有两种机制：推送机制（当内容变更后主动推送到 CDN 边缘节点）和拉取机制（先访问边缘节点，当没有内容时，回源到源服务器拿到内容并存储到节点上），两种方式各有利弊。使用 CDN 时要考虑 URL 的设计，比如 URL 中不能有随机数，否则每次都穿透 CDN 回源到源服务器，相当于 CDN 没有任何效果。对于爬虫，可以返回过期数据而选择不回源。

4. 接入层缓存

对于没有 CDN 缓存的应用来说，可以考虑使用如 Nginx 搭建一层接入层，该接入层可以考虑使用如下机制实现。

- **URL重写：**将URL按照指定的顺序或者格式重写，去除随机数。
- **一致性哈希：**按照指定的参数（如分类/商品编号）做一致性Hash，从而保证相同数据落到一台服务器上。
- **proxy_cache：**使用内存级/SSD级代理缓存来缓存内容。
- **proxy_cache_lock：**使用lock机制，将多个回源合并为一个，以减少回源量，并设置相应的lock超时时间。
- **shared_dict：**如果架构使用了nginx+lua实现，则可以考虑使用lua shared_dict进行cache，最大的好处就是reload缓存不会丢失。

此处要注意，对于托底（或兜底，指降级后显示的）数据或异常数据，不应该让其缓存，否则用户会在很长一段时间里看到这些数据。

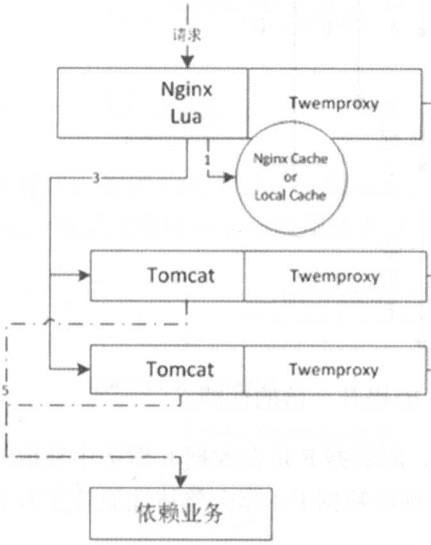
5. 应用层缓存

我们使用 Tomcat 时，可以使用堆内缓存/堆外缓存，堆内缓存的最大问题就是重启时内存中的缓存会丢失，此时流量风暴来临，则有可能冲垮应用；还可考虑使用 local redis cache 来代替堆外内存；或在接入层使用 shared_dict 来将缓存前置，以减少风暴。

local redis cache，通过在应用所在服务器上部署一组 Redis，应用直接读本机 Redis 获取数据，多机之间使用主从机制同步数据。这种方式没有网络消耗，性能是最优的。

6. 分布式缓存

有一种机制是要废弃分布式缓存，改成应用 local redis cache 情况下，如果数据量不大，这种架构是最优的。但是如果数据量太大，单服务器存储不了，那么可以使用分片机制将流量分散到多台，或者直接用分布式缓存实现。常见的分片规则就是一致性哈希了。



如上图所示就是我们一个应用的架构。

- 首先接入层（nginx+lua）读取本地proxy cache / local cache。
- 如果不命中，则接入层会接着读取分布式Redis集群。
- 如果还不命中，则会回源到Tomcat，然后读取Tomcat应用堆内cache。
- 如果缓存都没命中，则调用依赖业务来获取数据，然后异步化写到Redis集群。

因为我们使用了 nginx+lua，第二、三步时可使用 lua-resty-lock 非阻塞锁减少峰值时的回源量；如果你的服务是用户维度的，那么这种非阻塞锁大部分情况下不会有太大作用（要看具体场景）。

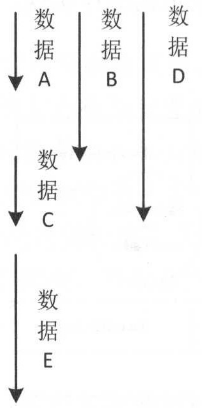
1.1.7 并发化

假设一个读服务需要如下数据。

目标数据	数据 A	数据 B	数据 C	数据 D	数据 E
获取时间	10ms	15ms	20ms	5ms	10ms

如果串行获取，那么需要 60ms。

而如果数据 C 依赖数据 A 和数据 B、数据 D 谁也不依赖、数据 E 依赖数据 C，那么我们可以这样来获取数据。



如果并发化获取，则需要 30ms，能提升一倍的性能。

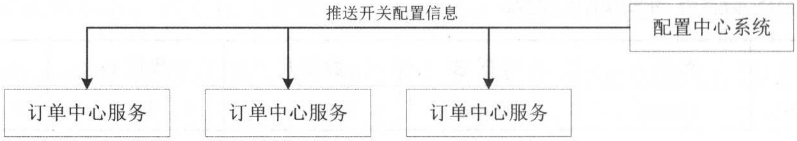
假设数据 E 还依赖数据 F（5ms），而数据 F 是在数据 E 服务中获取的，此时，就可以考虑在此服务中取数据 A/B/D 时，预取数据 F，那么整体性能就变为 25ms。

1.2 高可用原则

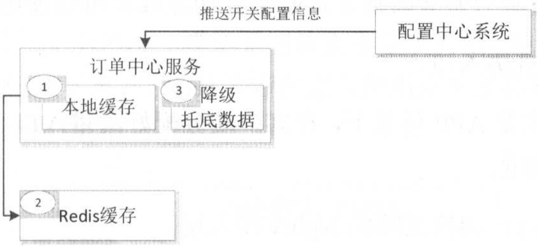
1.2.1 降级

对于一个高可用服务，很重要的一个设计就是降级开关，在设计降级开关时，主要依据如下思路。

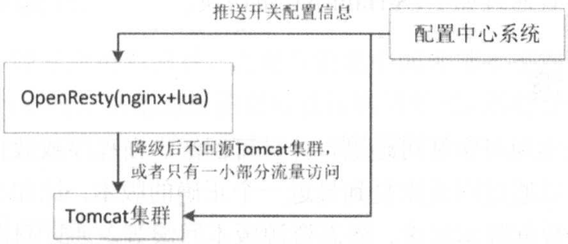
1. 开关集中化管理：通过推送机制把开关推送到各个应用。



2．可降级的多级读服务：比如服务调用降级为只读本地缓存、只读分布式缓存、只读默认降级数据（如库存状态默认有货）。



3．开关前置化：如架构是 Nginx→Tomcat，可以将开关前置到 Nginx 接入层，在 Nginx 层做开关，请求流量回源后端应用或者只是一小部分流量回源。



4．业务降级：当高并发流量来袭，在电商系统大促设计时保障用户能下单、能支付是核心要求，并保障数据最终一致性即可。这样就可以把一些同步调用改成异步调用，优先处理高优先级数据或特殊特征的数据，合理分配进入系统的流量，以保障系统可用。

1.2.2 限流

限流的目的是防止恶意请求流量、恶意攻击，或者防止流量超出系统峰值。可以考虑如下思路。

1．恶意请求流量只访问到 cache。

2．对于穿透到后端应用的流量可以考虑使用 Nginx 的 limit 模块处理。

3．对于恶意 IP 可以使用 nginx deny 进行屏蔽。

原则是限制流量穿透到后端薄弱的应用层。

1.2.3 切流量

对于一个大型应用，切流量是非常重要的，比如多机房环境下某个机房挂了，或者某个机架挂了，或者某台服务器挂了，都需要切流量，可以使用如下手段进行切换。

1．**DNS**：切换机房入口。

2．**HttpDNS**：主要 APP 场景下，在客户端分配好流量入口，绕过运营商 LocalDNS 并实现更精准流量调度。

3．**LVS/HaProxy**：切换故障的 Nginx 接入层。

4．**Nginx**：切换故障的应用层。

另外，有些应用为了更方便切换，还可以在 Nginx 接入层做切换，通过 Nginx 进行一些流量切换，而没有通过如 LVS/HaProxy 做切换。

1.2.4 可回滚

版本化的目的是实现可审计可追溯，并且可回滚。当程序或数据出错时，如果有版本化机制，那么就可以通过回滚恢复到最近一个正确的版本，比如事务回滚、代码库回滚、部署版本回滚、数据版本回滚、静态资源版本回滚等。通过回滚机制可保证系统某些场景下的高可用。

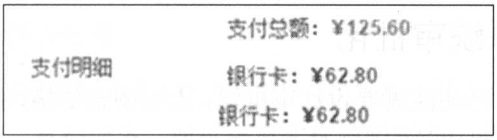
本书将介绍通过负载均衡和反向代理实现分流，通过限流保护服务免受雪崩之灾，通过降级实现部分可用、有损服务，通过隔离实现故障隔离，通过设置合理的超时与重试机制避免请求堆积造成雪崩，通过回滚机制快速修复错误版本。上述原则用来保护系统，往往能实现系统高可用。

1.3 业务设计原则

这些原则本书只进行简单介绍，不会展开讲解，大家可以自行研究学习。

1.3.1 防重设计

比如，结算页需要考虑重复提交，还有如下单扣减库存时需要防止重复扣减库存。解决方案可以考虑防重 key、防重表。而有些场景如重复支付，是因为有的电商网站同时支持微信支付、京东支付，渠道不一样是无法防止重复支付的。但是，在系统设计时，需要将支付的每笔情况记录下来。下图是笔者在京东使用京东支付和微信支付模拟的重复支付之后进行退款的支付明细。



1.3.2 幂等设计

在交易系统中，经常会用到消息，而现有消息中间件基本不保证不发生重复消息的消费。因此，需要业务系统在重复消息消费时进行幂等处理。还有在使用第三方支付时，第三方支付会进行异步回调，也要考虑做好回调的幂等处理。

1.3.3 流程可定义

如果接触过保险业务，就会发现不同保险的理赔服务是不一样的。我们在系统设计时就设计了一套理赔流程服务。而承保流程和理赔流程是分离的，在需要时进行关联，从而可以复用一些理赔流程，并提供个性化的理赔流程。

1.3.4 状态与状态机

在设计交易订单系统时，会存在正向状态（待付款、待发货、已发货、完成）和逆向状态（取消、退款）等，正向状态和逆向状态应该根据系统的特征来决定要不要分离存储。状态设计时应有状态轨迹，方便用户跟踪当前订单的轨迹并记录相关日志，万一出问题时可回溯问题。

另外，还有订单状态的变迁，比如待支付、已支付待发货、待收货、完成的迁移。要考虑要不要使用状态机来驱动状态的变更和后续流程节点操作，尤其当状态很多的时候使用状态机能更好地控制状态迁移。

还要考虑并发状态修改问题，如一个订单同时只能有一个修改；状态变更的有序问

题，以及状态变更消息的先到后到问题，如支付成功消息和用户取消消息的时间差。

1.3.5 后台系统操作可反馈

在笔者接触过的系统中，很多场景都需要反馈，比如，修改了某些内容后想预览看看最终效果，即想得到一些反馈；还有一些是在规则系统中，希望看到这些规则在系统数据下的反馈。因此，在设计后台系统时，需考虑效果的可预览、可反馈。

1.3.6 后台系统审批化

对于有些重要的后台功能需要设计审批流，比如调整价格，并对操作进行日志记录，从而保证操作可追溯、可审计。

1.3.7 文档和注释

笔者接触的一些系统是完全没有文档、代码没有注释的，完全是人传人。这将导致后来人接手很痛苦，而且对有些代码是完全不敢改动的，比如，有些代码完全是因为业务的一些特殊情况而写的，可以说没有注释是完全不懂为什么那么做的。因此，在一个系统发展的一开始就应该有文档库（设计架构、设计思想、数据字典/业务流程、现有问题），业务代码和特殊需求都要有注释。

1.3.8 备份

包括代码和人员。代码主要提交到代码仓库进行管理和备份，代码仓库应该至少具备多版本的功能。人员备份指的是一个系统至少应该有两名开发人员是了解的，即使其中一名离职了也不会出现新人接手之后手忙脚乱事故频发的状况。还有一些是“核心人员”，写着系统的核心代码，被认为是“不可替代的”，这种情况也是尽可能地让他带一名兄弟一起开发核心代码（业务系统），即使离职也还是可以努力一下克服困难。

1.4 总结

对于一个系统设计来说，不仅需要考虑实现业务功能，还要保证系统高并发、高可

用、高可靠等。在系统容量规划（流量、容量等）、SLA 制定（吞吐量、响应时间、可用性、降级方案等）、压测方案（线上、线上等）、监控报警（机器负载、响应时间、可用率等）、应急预案（容灾、降级、限流、隔离、切流量、可回滚等）等方面，也要有一些原则来指导大家。其中，每一个方向都是很复杂的，为了能讲解地较为深入，本书将从高并发和高可用两个方面来讲解，并配合案例实战使读者能参考案例，来理解这些原则并解决系统痛点。

本书将介绍缓存、异步并发、连接池、线程池、如何扩容、消息队列、分布式任务等高并发原则来提升系统吞吐量。

本书将介绍通过负载均衡和反向代理实现分流，通过限流保护服务免受雪崩之灾，通过降级实现部分可用、有损服务，通过隔离实现故障隔离，通过设置合理的超时与重试机制避免请求堆积造成雪崩，通过回滚机制快速修复错误版本；通过上述原则来保护系统，使得系统高可用。

缓存回收策略：空间/容量/时间
缓存回收算法
FIFO/LRU/LFU
Java堆/Java堆外/磁盘缓存
Guava/Ehcache/MapDB
缓存使用模式
Cache-Aside/Cache-As-SoR/Copy Pattern
应用级缓存
浏览器缓存
HttpClient客户端缓存
Nginx代理层缓存
HTTP缓存
分布式缓存
热点数据与更新缓存
更新缓存与原子性
缓存崩溃与快速修复
多级缓存
数据库连接池
HttpClient连接池
线程池
池化
高并发
异步并发
同步阻塞调用
异步Future
异步Callback
异步编排CompletableFuture
请求缓存
请求合并
扩容
单体应用垂直扩容
单体应用水平扩容
应用拆分
数据库拆分:水平/垂直拆分
使用sharding-jdbc
分库分表/读写分离
数据异构
任务系统扩容（Elastic-Job）
队列
异步处理/系统解耦
数据同步/流量削峰
缓冲队列/任务队列/消息队列
请求队列/数据总线队列
Disruptor+Redis队列
基于Canal实现数据异构

第 2 部分　高可用

- 负载均衡与反向代理
- 隔离术
- 限流详解
- 降级特技
- 超时与重试机制
- 回滚机制
- 压测与预案

2 负载均衡与反向代理

当我们的应用单实例不能支撑用户请求时，此时就需要扩容，从一台服务器扩容到两台、几十台、几百台。然而，用户访问时是通过如 http://www.jd.com 的方式访问，在请求时，浏览器首先会查询 DNS 服务器获取对应的 IP，然后通过此 IP 访问对应的服务。

因此，一种方式是 www.jd.com 域名映射多个 IP，但是，存在一个最简单的问题，假设某台服务器重启或者出现故障，DNS 会有一定的缓存时间，故障后切换时间长，而且没有对后端服务进行心跳检查和失败重试的机制。

因此，外网 DNS 应该用来实现用 GSLB（全局负载均衡）进行流量调度，如将用户分配到离他最近的服务器上以提升体验。而且当某一区域的机房出现问题时（如被挖断了光缆），可以通过 DNS 指向其他区域的 IP 来使服务可用。

可以在站长之家使用“DNS 查询”，查询 c.3.cn 可以看到类似如下的结果。

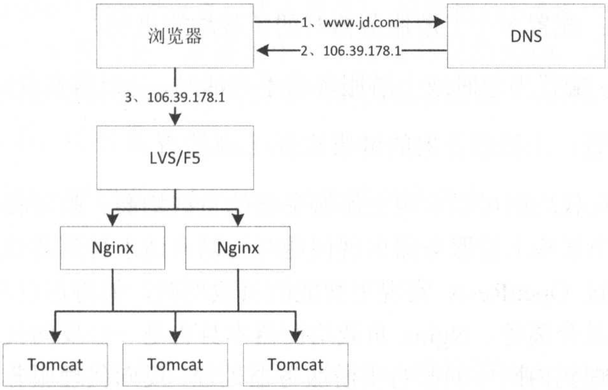
北京[电信]	106.39.164.153 [北京市 北京电信互联网数据中心]	98
上海[电信]	106.39.164.153 [北京市 北京电信互联网数据中心]	116
西藏[移动]	111.13.28.153 [北京市 移动]	19
天津[移动]	-	-
湖北[移动]	106.39.164.153 [北京市 北京电信互联网数据中心]	600

即不同的运营商返回的公网 IP 是不一样的。

对于内网 DNS，可以实现简单的轮询负载均衡。但是，还是那句话，会有一定的缓

存时间并且没有失败重试机制。因此，我们可以考虑选择如 HaProxy 和 Nginx。

而对于一般应用来说，有 Nginx 就可以了。但 Nginx 一般用于七层负载均衡，其吞吐量是有一定限制的。为了提升整体吞吐量，会在 DNS 和 Nginx 之间引入接入层，如使用 LVS（软件负载均衡器）、F5（硬负载均衡器）可以做四层负载均衡，即首先 DNS 解析到 LVS/F5，然后 LVS/F5 转发给 Nginx，再由 Nginx 转发给后端 Real Server。



对于一般业务开发人员来说，我们只需要关心到 Nginx 层面就够了，LVS/F5 一般由系统/运维工程师来维护。Nginx 目前提供了 HTTP（ngx_http_upstream_module）七层负载均衡，而 1.9.0 版本也开始支持 TCP（ngx_stream_upstream_module）四层负载均衡。

此处再澄清几个概念。二层负载均衡是通过改写报文的目标 MAC 地址为上游服务器 MAC 地址，源 IP 地址和目标 IP 地址是没有变的，负载均衡服务器和真实服务器共享同一个 VIP，如 LVS DR 工作模式。四层负载均衡是根据端口将报文转发到上游服务器（不同的 IP 地址+端口），如 LVS NAT 模式、HaProxy，七层负载均衡是根据端口号和应用层协议如 HTTP 协议的主机名、URL，转发报文到上游服务器（不同的 IP 地址+端口），如 HaProxy、Nginx。

这里再介绍一下 LVS DR 工作模式，其工作在数据链路层，LVS 和上游服务器共享同一个 VIP，通过改写报文的目标 MAC 地址为上游服务器 MAC 地址实现负载均衡，上游服务器直接响应报文到客户端，不经过 LVS，从而提升性能。但因为 LVS 和上游服务器必须在同一个子网，为了解决跨子网问题而又不影响负载性能，可以选择在 LVS 后边挂 HaProxy，通过四到七层负载均衡器 HaProxy 集群来解决跨网和性能问题。这两个“半成品”的东西相互取长补短，组合起来就变成了一个“完整”的负载均衡器。现在 Nginx 的 stream 也支持 TCP，所以 Nginx 也算是一个四到七层的负载均衡器，一般场景下可以用 Nginx 取代 HaProxy。

在继续讲解之前，首先统一几个术语。接入层、反向代理服务器、负载均衡服务器，在本文中如无特殊说明则指的是 Nginx。upstream server 即上游服务器，指 Nginx 负载均衡到的处理业务的服务器，也可以称之为 real server，即真实处理业务的服务器。

对于负载均衡我们要关心的几个方面如下。

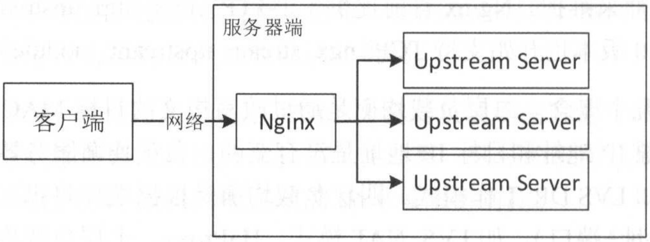
上游服务器配置：使用 upstream server 配置上游服务器。

负载均衡算法：配置多个上游服务器时的负载均衡机制。

失败重试机制：配置当超时或上游服务器不存活时，是否需要重试其他上游服务器。

服务器心跳检查：上游服务器的健康检查/心跳检查。

Nginx 提供的负载均衡可以实现上游服务器的负载均衡、故障转移、失败重试、容错、健康检查等，当某些上游服务器出现问题时可以将请求转到其他上游服务器以保障高可用，并可以通过 OpenResty 实现更智能的负载均衡，如将热点与非热点流量分离、正常流量与爬虫流量分离等。Nginx 负载均衡器本身也是一台反向代理服务器，将用户请求通过 Nginx 代理到内网中的某台上游服务器处理，反向代理服务器可以对响应结果进行缓存、压缩等处理以提升性能。Nginx 作为负载均衡器/反向代理服务器如下图所示。



本章首先会讲解 Nginx HTTP 负载均衡，最后会讲解使用 Nginx 实现四层负载均衡。

2.1 upstream 配置

第一步我们需要给 Nginx 配置上游服务器，即负载均衡到的真实处理业务的服务器，通过在 http 指令下配置 upstream 即可。

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;
}
```

upstream server 的主要配置如下。

- **IP地址和端口**：配置上游服务器的IP地址和端口。
- **权重**：weight用来配置权重，默认都是1，权重越高分配给这台服务器的请求就越多（如上配置为每三次请求中一个请求转发给9080，其余两个请求转发给9090），需要根据服务器的实际处理能力设置权重（比如，物理服务器和虚拟机就需要不同的权重）。

然后，我们可以配置如下 proxy_pass 来处理用户请求。

```
location / {
    proxy_pass http://backend;
}
```

当访问 Nginx 时，会将请求反向代理到 backend 配置的 upstream server。接下来我们看一下负载均衡算法。

2.2 负载均衡算法

负载均衡用来解决用户请求到来时如何选择 upstream server 进行处理，默认采用的是 round-robin（轮询），同时支持其他几种算法。

- **round-robin**：轮询，默认负载均衡算法，即以轮询的方式将请求转发到上游服务器，通过配合weight配置可以实现基于权重的轮询。
- **ip_hash**：根据客户IP进行负载均衡，即相同的IP将负载均衡到同一个upstream server。

```
upstream backend {
    ip_hash;
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;
}
```

- **hash key [consistent]**：对某一个key进行哈希或者使用一致性哈希算法进行负载均衡。使用Hash算法存在的问题是，当添加/删除一台服务器时，将导致很多key被重新负载均衡到不同的服务器（从而导致后端可能出现问题）；因此，建议考虑使用一致性哈希算法，这样当添加/删除一台服务器时，只有少数key将被重新负载均衡到不同的服务器。
  - **哈希算法**：此处是根据请求 uri 进行负载均衡，可以使用 Nginx 变量，因此，可以实现复杂的算法。

```
upstream backend {
   hash $uri;
   server 192.168.61.1:9080  weight=1;
   server 192.168.61.1:9090  weight=2;
}
```

  - **一致性哈希算法**：consistent_key 动态指定。

```
upstream nginx_local_server {
   hash $consistent_key consistent;
   server 192.168.61.1:9080  weight=1;
   server 192.168.61.1:9090  weight=2;
}
```

如下 location 指定了一致性哈希 key，此处会优先考虑请求参数 cat（类目），如果没有，则再根据请求 uri 进行负载均衡。

```
location / {
   set $consistent_key $arg_cat;
   if ($consistent_key = "") {
      set $consistent_key $request_uri;
   }
}
```

而实际我们是通过 Lua 设置一致性哈希 key。

```
set_by_lua_file $consistent_key "lua_balancing.lua";

lua_balancing.lua 代码。
local consistent_key = args.cat
if not consistent_key or consistent_key == '' then
   consistent_key = ngx_var.request_uri
end

local value = balancing_cache:get(consistent_key)
if not value then
```

```
    success, err = balancing_cache:set(consistent_key, 1, 60)
else
    newval, err = balancing_cache:incr(consistent_key, 1)
end
```

如果某一个分类请求量太大，上游服务器可能处理不了这么多的请求，此时可以在一致性哈希 key 后加上递增的计数以实现类似轮询的算法。

```
if newval > 5000 then
    consistent_key = consistent_key .. '_' .. newval
end
```

- **least_conn**：将请求负载均衡到最少活跃连接的上游服务器。如果配置的服务器较少，则将转而使用基于权重的轮询算法。

Nginx 商业版还提供了 least_time，即基于最小平均响应时间进行负载均衡。

2.3 失败重试

主要有两部分配置：upstream server 和 proxy_pass。

```
upstream backend {
    server 192.168.61.1:9080 max_fails=2 fail_timeout=10s weight=1;
    server 192.168.61.1:9090 max_fails=2 fail_timeout=10s weight=1;
}
```

通过配置上游服务器的 max_fails 和 fail_timeout，来指定每个上游服务器，当 fail_timeout 时间内失败了 max_fails 次请求，则认为该上游服务器不可用/不存活，然后将摘掉该上游服务器，fail_timeout 时间后会再次将该服务器加入到存活上游服务器列表进行重试。

```
location /test {
    proxy_connect_timeout 5s;
    proxy_read_timeout 5s;
    proxy_send_timeout 5s;

    proxy_next_upstream error timeout;
    proxy_next_upstream_timeout 10s;
    proxy_next_upstream_tries 2;
```

```
    proxy_pass http://backend;
    add_header upstream_addr $upstream_addr;
}
```

然后进行 proxy_next_upstream 相关配置，当遇到配置的错误时，会重试下一台上游服务器。

详细配置请参考第 6 章中代理层超时与重试机制的 Nginx 部分。

2.4 健康检查

Nginx 对上游服务器的健康检查默认采用的是惰性策略，Nginx 商业版提供了 health_check 进行主动健康检查。当然也可以集成 nginx_upstream_check_module（https://github.com/yaoweibin/nginx_upstream_check_module）模块来进行主动健康检查。

nginx_upstream_check_module 支持 TCP 心跳和 HTTP 心跳来实现健康检查。

2.4.1 TCP 心跳检查

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;
    check interval=3000 rise=1 fall=3 timeout=2000 type=tcp;
}
```

此处配置使用 TCP 进行心跳检测。

- **interval**：检测间隔时间，此处配置了每隔3s检测一次。
- **fall**：检测失败多少次后，上游服务器被标识为不存活。
- **rise**：检测成功多少次后，上游服务器被标识为存活，并可以处理请求。
- **timeout**：检测请求超时时间配置。

2.4.2 HTTP 心跳检查

```
upstream backend {
```

```
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2;

    check interval=3000 rise=1 fall=3 timeout=2000 type=http;
    check_http_send "HEAD /status HTTP/1.0\r\n\r\n";
    check_http_expect_alive http_2xx http_3xx;
}
```

HTTP 心跳检查有如下两个需要额外配置。

- **check_http_send**：即检查时发的HTTP请求内容。
- **check_http_expect_alive**：当上游服务器返回匹配的响应状态码时，则认为上游服务器存活。

此处需要注意，检查间隔时间不能太短，否则可能因为心跳检查包太多造成上游服务器挂掉，同时要设置合理的超时时间。

本文使用的是 openresty/1.11.2.1（对应 nginx-1.11.2），安装 Nginx 之前需要先打 nginx_upstream_check_module 补丁（check_1.9.2+.patch），到 Nginx 目录下执行如下 shell：

patch -p0 < /usr/servers/nginx_upstream_check_module-master/check_1.9.2+.patch。

如果不安装补丁，那么 nginx_upstream_check_module 模块是不工作的，建议使用 wireshark 抓包查看其是否工作。

2.5 其他配置

2.5.1 域名上游服务器

```
upstream backend {
    server c0.3.cn;
    server c1.3.cn;
}
```

在 Nginx 社区版中，是在 Nginx 解析配置文件的阶段将域名解析成 IP 地址并记录到 upstream 上，当这两个域名对应的 IP 地址发生变化时，该 upstream 不会更新。Nginx 商业版才支持动态更新。

不过，proxy_pass http://c0.3.cn 是支持动态域名解析的。

2.5.2 备份上游服务器

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2 backup;
}
```

将 9090 端口上游服务器配置为备上游服务器，当所有主上游服务器都不存活时，请求会转发给备上游服务器。

如通过缩容上游服务器进行压测时，要摘掉一些上游服务器进行压测，但为了保险起见会配置一些备上游服务器，当压测的上游服务器都挂掉时，流量可以转发到备上游服务器，从而不影响用户请求处理。

2.5.3 不可用上游服务器

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2 down;
}
```

9090 端口上游服务器配置为永久不可用，当测试或者机器出现故障时，暂时通过该配置临时摘掉机器。

2.6 长连接

此处只涉及如何配置 Nginx 与上游服务器的长连接，而客户端与 Nginx 之间的长连接可以参考位置第 6 章的相应部分。

可以通过 keepalive 指令配置长连接数量。

```
upstream backend {
    server 192.168.61.1:9080  weight=1;
    server 192.168.61.1:9090  weight=2 backup;
    keepalive 100;
```

```
}
```

通过该指令配置了每个 Worker 进程与上游服务器可缓存的空闲连接的最大数量。当超出这个数量时，最近最少使用的连接将被关闭。keepalive 指令不限制 Worker 进程与上游服务器的总连接。

如果想要跟上游服务器建立长连接，则一定别忘了以下配置。

```
location / {
    #支持 keep-alive
    proxy_http_version 1.1;
    proxy_set_header Connection "";
    proxy_pass http://backend;
}
```

如果是 http/1.0，则需要配置发送“Connection: Keep-Alive”请求头。

上游服务器不要忘记开启长连接支持。

接下来，我们看一下 Nginx 是如何实现 keepalive 的（ngx_http_upstream_keepalive_module），获取连接时的部分代码。

```
ngx_http_upstream_get_keepalive_peer(ngx_peer_connection_t *pc,
void *data) {
    //1. 首先询问负载均衡使用哪台服务器（IP 和端口）
    rc = kp->original_get_peer(pc, kp->data);

    cache = &kp->conf->cache;
    //2. 轮询 “空闲连接池”
    for (q = ngx_queue_head(cache);
        q != ngx_queue_sentinel(cache);
         q = ngx_queue_next(q))
    {
        item = ngx_queue_data(q, ngx_http_upstream_keepalive_cache_t, queue);
        c = item->connection;
        //2.1. 如果“空闲连接池”缓存的连接 IP 和端口与负载均衡到的 IP 和端口相同，
        //则使用此连接
        if (ngx_memn2cmp((u_char *) &item->sockaddr, (u_char *) pc->sockaddr,
                         item->socklen, pc->socklen) == 0) {
            //2.2 从“空闲连接池”移除此连接并压入“释放连接池”栈顶
            ngx_queue_remove(q);
            ngx_queue_insert_head(&kp->conf->free, q);

            goto found;
        }
```

```
    }
    //3. 如果“空闲连接池”没有可用的长连接，将创建短连接
    return NGX_OK;
```

释放连接时的部分代码如下。

```
ngx_http_upstream_free_keepalive_peer(ngx_peer_connection_t *pc,
void *data, ngx_uint_t state) {
    c = pc->connection;//当前要释放的连接
    //1. 如果“释放连接池”没有待释放连接，那么需要从“空闲连接池”腾出一个空间给新
    //的连接使用（这种情况存在于创建连接数超出了连接池大小时，这就会出现震荡）
    if (ngx_queue_empty(&kp->conf->free)) {
        q = ngx_queue_last(&kp->conf->cache);
        ngx_queue_remove(q);
        item  =  ngx_queue_data(q,  ngx_http_upstream_keepalive_cache_t,
queue);
        ngx_http_upstream_keepalive_close(item->connection);
    } else {//2. 从“释放连接池”释放一个连接
        q = ngx_queue_head(&kp->conf->free);
        ngx_queue_remove(q);
        item = ngx_queue_data(q, ngx_http_upstream_keepalive_cache_t,
queue);
    }
    //3. 将当前连接压入“空闲连接池”栈顶供下次使用
    ngx_queue_insert_head(&kp->conf->cache, q);
    item->connection = c;
```

总长连接数是“空闲连接池”+“释放连接池”的长连接总数。首先，长连接配置不会限制 Worker 进程可以打开的总连接数（超了的作为短连接）。另外，连接池一定要根据实际场景合理进行设置。

1. 空闲连接池太小，连接不够用，需要不断建连接。

2. 空闲连接池太大，空闲连接太多，还没使用就超时。

另外，建议只对小报文开启长连接。

2.7 HTTP 反向代理示例

反向代理除了实现负载均衡之外，还提供如缓存来减少上游服务器的压力。

1. 全局配置（proxy cache）

```
proxy_buffering              on;
proxy_buffer_size            4k;
proxy_buffers                512 4k;
proxy_busy_buffers_size      64k;
proxy_temp_file_write_size   256k;
proxy_cache_lock             on;
proxy_cache_lock_timeout     200ms;
proxy_temp_path              /tmpfs/proxy_temp;
proxy_cache_path             /tmpfs/proxy_cache levels=1:2 keys_zone
                             =cache:512m inactive=5m max_size=8g;
proxy_connect_timeout  3s;
proxy_read_timeout    5s;
proxy_send_timeout    5s;
```

开启 proxy buffer，缓存内容将存放在 tmpfs（内存文件系统）以提升性能，设置超时时间。

2. location 配置

```
location ~ ^/backend/(.*)$ {
    #设置一致性哈希负载均衡 key
    set_by_lua_file $consistent_key "/export/App/c.3.cn/lua/lua_
balancing_backend.properties";
    #失败重试配置
    proxy_next_upstream error timeout http_500 http_502 http_504;
    proxy_next_upstream_timeout 2s;
    proxy_next_upstream_tries 2;

    #请求上游服务器使用 GET 方法（不管请求是什么方法）
    proxy_method GET;
    #不给上游服务器传递请求体
    proxy_pass_request_body off;
```

```
    #不给上游服务器传递请求头
    proxy_pass_request_headers off;
    #设置上游服务器的哪些响应头不发送给客户端
    proxy_hide_header Vary;
    #支持 keep-alive
    proxy_http_version 1.1;
    proxy_set_header Connection "";
    #给上游服务器传递 Referer、Cookie 和 Host（按需传递）
    proxy_set_header Referer $http_referer;
    proxy_set_header Cookie $http_cookie;
    proxy_set_header Host web.c.3.local;
    proxy_pass http://backend /$1$is_args$args;
}
```

我们开启了 proxy_pass_request_body 和 proxy_pass_request_headers，禁止向上游服务器传递请求头和内容体，从而使得上游服务器不受请求头攻击，也不需要解析；如果需要传递，则使用 proxy_set_header 按需传递即可。

我们还可以通过如下配置来开启 gzip 支持，减少网络传输的数据包大小。

```
gzip                on;
gzip_min_length     1k;
gzip_buffers        16 16k;
gzip_http_version   1.0;
gzip_proxied        any;
gzip_comp_level     2;
gzip_types          text/plain application/x-javascript text/css
                    application/xml;
gzip_vary           on;
```

对于内容型响应建议开启 gzip 压缩，gzip_comp_level 压缩级别要根据实际压测来决定（带宽和吞吐量之间的抉择）。

2.8 HTTP 动态负载均衡

如上的负载均衡实现中，每次 upstream 列表有变更，都需要到服务器进行修改，首先是管理容易出现问题，而且对于 upstream 服务上线无法自动注册到 nginx upstream 列

表。因此，我们需要一种服务注册，可以将 upstream 动态注册到 Nginx 上，从而实现 upstream 服务的自动发现。

Consul 是一款开源的分布式服务注册与发现系统，通过 HTTP API 可以使得服务注册、发现实现起来非常简单，它支持如下特性。

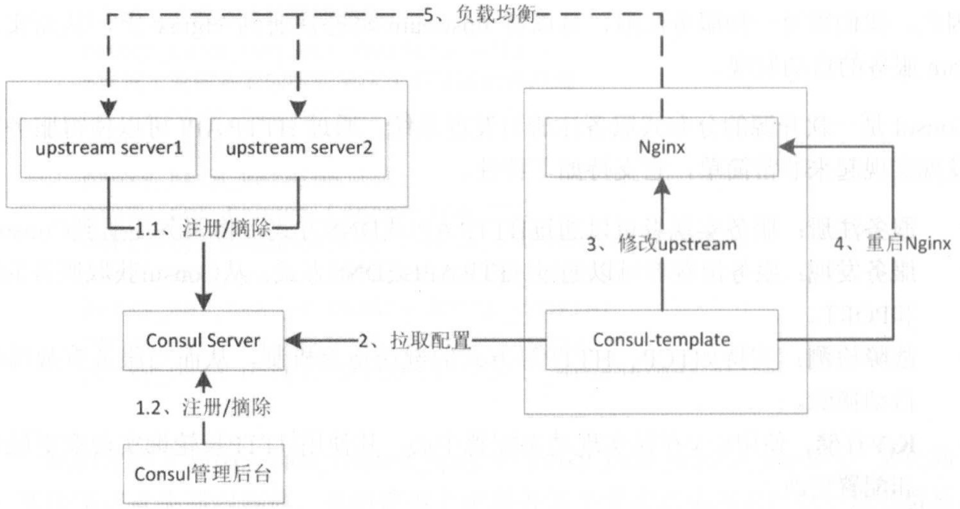
- **服务注册：**服务实现者可以通过HTTP API或DNS方式，将服务注册到Consul。
- **服务发现：**服务消费者可以通过HTTP API或DNS方式，从Consul获取服务的IP和PORT。
- **故障检测：**支持如TCP、HTTP等方式的健康检查机制，从而当服务有故障时自动摘除。
- **K/V存储：**使用K/V存储实现动态配置中心，其使用HTTP长轮询实现变更触发和配置更改。
- **多数据中心：**支持多数据中心，可以按照数据中心注册和发现服务，即支持只消费本地机房服务，使用多数据中心集群还可以避免单数据中心的单点故障。
- **Raft算法：**Consul使用Raft算法实现集群数据一致性。

通过 Consul 可以管理服务注册与发现，接下来需要有一个与 Nginx 部署在同一台机器的 Agent 来实现 Nginx 配置更改和 Nginx 重启功能。我们有 Confd 或者 Consul-template 两个选择，而 Consul-template 是 Consul 官方提供的，我们就选择它了。其使用 HTTP 长轮询实现变更触发和配置更改（使用 Consul 的 watch 命令实现）。也就是说，我们使用 Consul-template 实现配置模板，然后拉取 Consul 配置渲染模板来生成 Nginx 实际配置。

除 Consul 外，还有一个选择是 etcd3，其使用了 gRPC 和 protobuf 可以说是一个亮点。不过，etcd3 目前没有提供多数据中心、故障检测、Web 管理界面。

2.8.1 Consul+Consul-template

接下来，让我们看一下如何来实现 Nginx 动态配置。首先，下图是我们要实现的架构图。



首先，upstream 服务启动，我们通过管理后台向 Consule 注册服务。

我们需要在 Nginx 机器上部署并启动 Consul-template Agent，其通过长轮询监听服务变更。

Consul-template 监听到变更后，动态修改 upstream 列表。

Consul-template 修改完 upstream 列表后，调用重启 Nginx 脚本重启 Nginx。

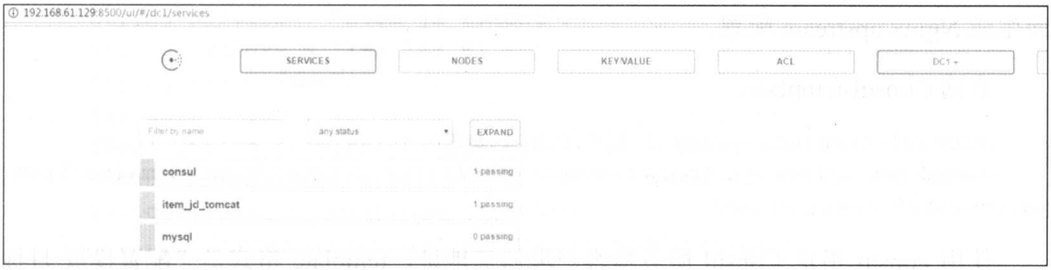
整个实现过程还是比较简单的，不过，实际生产环境要复杂得多。我们使用了 Consul 0.7.0 和 Consul-template 0.16.0 来实现。

1．Consul-Server

首先我们要启动 Consul-Server。

```
./consul agent -server -bootstrap-expect 1 -data-dir /tmp/consul
-bind 0.0.0.0 -client 0.0.0.0
```

此处需要使用 data-dir 指定 Agent 状态存储位置，bind 指定集群通信的地址，client 指定客户端通信的地址（如 Consul-template 与 Consul 通信）。在启动时还可以使用 -ui-dir ./ui/指定 Consul Web UI 目录，实现通过 Web UI 管理 Consul，然后访问如 http://127.0.0.1:8500 即可看到控制界面。



使用如下 HTTP API 注册服务。

```
curl -X PUT http://127.0.0.1:8500/v1/catalog/register -d '{"Datacenter": "dc1",
"Node": "tomcat", "Address": "192.168.1.1","Service": {"Id" : "192.168.1.1:8080",
"Service": "item_jd_tomcat", "tags": ["dev"], "Port": 8080}}'
curl -X PUT http://127.0.0.1:8500/v1/catalog/register -d '{"Datacenter": "dc1",
"Node": "tomcat", "Address": "192.168.1.2","Service": {"Id" : "192.168.1.1:8090",
"Service": "item_jd_tomcat", "tags": ["dev"], "Port": 8090}}'
```

Datacenter 指定数据中心，Address 指定服务 IP，Service.Id 指定服务唯一标识，Service.Service 指定服务分组，Service.tags 指定服务标签（如测试环境、预发环境等），Service.Port 指定服务器端口。

通过如下 HTTP API 摘除服务。

```
curl -X PUT http://127.0.0.1:8500/v1/catalog/deregister -d '{"Datacenter":
"dc1", "Node": "tomcat", "ServiceID" : "192.168.1.1:8080"}'
```

通过如下 HTTP API 发现服务。

```
curl http://127.0.0.1:8500/v1/catalog/service/item_jd_tomcat
```

可以看到，通过这几个 HTTP API 可以实现服务注册与发现。更多 API 请参考 https://www.consul.io/docs/agent/http.html。

2. Consul-template

接下来我们需要在 Consul-template 机器上添加一份配置模板 item.jd.tomcat.ctmpl。

```
upstream item_jd_tomcat {
    server 127.0.0.1:1111; #占位 server，必须有一个 server，否则无法启动
    {{range service "dev.item_jd_tomcat@dc1"}}
        server {{.Address}}:{{.Port}} weight=1;
    {{end}}
}
```

service 指定格式为：标签.服务@数据中心，然后通过循环输出 Address 和 Port，从

而生成 Nginx upstream 配置。

启动 Consul-template。

```
./consul-template -consul 127.0.0.1:8500 \
-template ./item.jd.tomcat.ctmpl:/usr/servers/nginx/conf/domains/item.
jd.tomcat:"./restart.sh"
```

使用 consul 指定 Consul 服务器客户端通信地址，template 格式是“配置模板:目标配置文件:脚本”，即通过配置模板更新目标配置文件，然后调用脚本重启 Nginx。

直接通过 Nginx include 指令将/usr/servers/nginx/conf/domains/item.jd.tomcat 包含到 nginx.conf 配置文件即可，restart.sh 脚本代码如下所示。

```
#!/bin/bash
ps -fe|grep nginx |grep -v grep
if [ $? -ne 0 ]
then
  sudo /usr/servers/nginx/sbin/nginx
  echo "nginx start"
else
  sudo /usr/servers/nginx/sbin/nginx -s reload
  echo "nginx reload"
fi
```

即如果 Nginx 没有启动，则启动，否则重启。

3. Java 服务

建议配合 Spring Boot+Consul Java Client 实现，我们使用的 Consul Java Client 如下。

```
<dependency>
    <groupId>com.orbitz.consul</groupId>
    <artifactId>consul-client</artifactId>
    <version>0.12.8</version>
</dependency>
```

如下代码是进行服务注册与摘除。

```
public static void main(String[] args) {
    //启动嵌入容器（如 Tomcat）
    SpringApplication.run(Bootstrap.class, args);
    //服务注册
    Consul consul = Consul.builder().withHostAndPort(HostAndPort. fromString
("192.168.61.129:8500")).build();
    final AgentClient agentClient = consul.agentClient();
```

```
    String service = "item_jd_tomcat";
    String address = "192.168.61.1";
    String tag = "dev";
    int port = 9080;
    final String serviceId = address + ":" + port;
    ImmutableRegistration.Builder builder = ImmutableRegistration.builder();
    builder.id(serviceId).name(service)
            .address(address).port(port).addTags(tag);
    agentClient.register(builder.build());
    //JVM 停止时摘除服务
    Runtime.getRuntime().addShutdownHook(new Thread() {
       @Override
       public void run() {
          agentClient.deregister(serviceId);
       }
    });
}
```

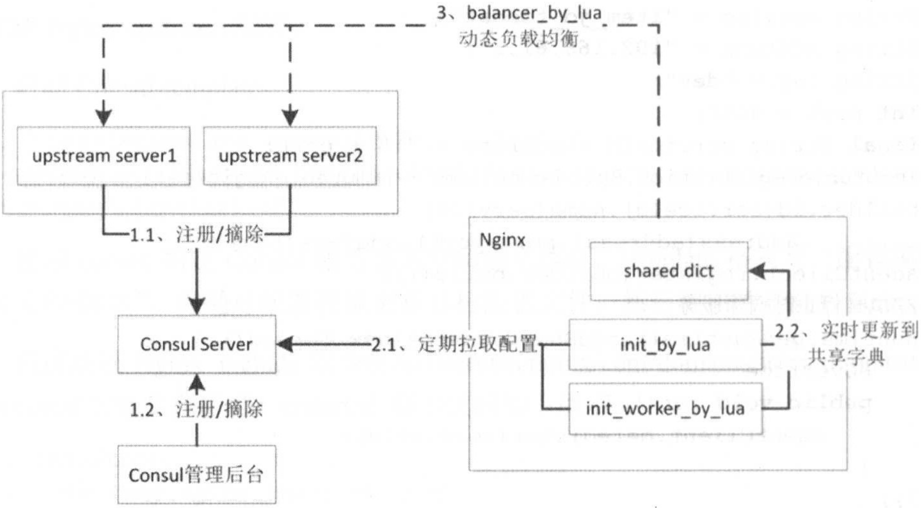
在 Spring Boot 启动后进行服务注册，然后在 JVM 停止时进行服务摘除。

到此我们就实现了动态 upstream 负载均衡，upstream 服务启动后自动注册到 Nginx，upstream 服务停止时，自动从 Nginx 上摘除。

通过 Consul+Consul-template 方式，每次发现配置变更都需要 reload nginx，而 reload 是有一定损耗的。而且，如果你需要长连接支持的话，那么当 reload nginx 时长连接所在 worker 进程会进行优雅退出，并当该 worker 进程上的所有连接都释放时，进程才真正退出（表现为 worker 进程处于 worker process is shutting down）。因此，如果能做到不 reload 就能动态更改 upstream，那么就完美了。对于社区版 Nginx 目前有三个选择：Tengine 的 Dyups 模块、微博的 Upsync 和使用 OpenResty 的 balancer_by_lua。微博使用 Upsync+Consul 实现动态负载均衡，而又拍云使用其开源的 slardar（Consul + balancer_by_lua）实现动态负载均衡。

2.8.2 Consul+OpenResty

使用 Consul 注册服务，使用 OpenResty balancer_by_lua 实现无 reload 动态负载均衡，架构如下所示。



1. 通过 upstream server 启动/停止时注册服务，或者通过 Consul 管理后台注册服务。

2. Nginx 启动时会调用 init_by_lua，启动时拉取配置，并更新到共享字典来存储 upstream 列表；然后通过 init_worker_by_lua 启动定时器，定期去 Consul 拉取配置并实时更新到共享字典。

3. balancer_by_lua 使用共享字典存储的 upstream 列表进行动态负载均衡。

dyna_upstreams.lua 模块

```
local http = require("socket.http")
local ltn12 = require("ltn12")
local cjson = require "cjson"
local function update_upstreams()
  local resp = {}
  http.request{
    url=
      "http://192.168.61.129:8500/v1/catalog/service/item_jd_tomcat",
    sink = ltn12.sink.table(resp)
  }
  resp = table.concat(resp)
  resp = cjson.decode(resp)

  local upstreams = {{ip="127.0.0.1", port=1111}}
  for i, v in ipairs(resp) do
    upstreams[i+1] = {ip=v.Address, port=v.ServicePort}
  end
```

```
  ngx.shared.upstream_list:set("item_jd_tomcat", cjson.encode (upstreams))
end

local function get_upstreams()
   local upstreams_str = ngx.shared.upstream_list:get("item_jd_tomcat")
end


local _M = {
   update_upstreams = update_upstreams,
   get_upstreams = get_upstreams
}
```

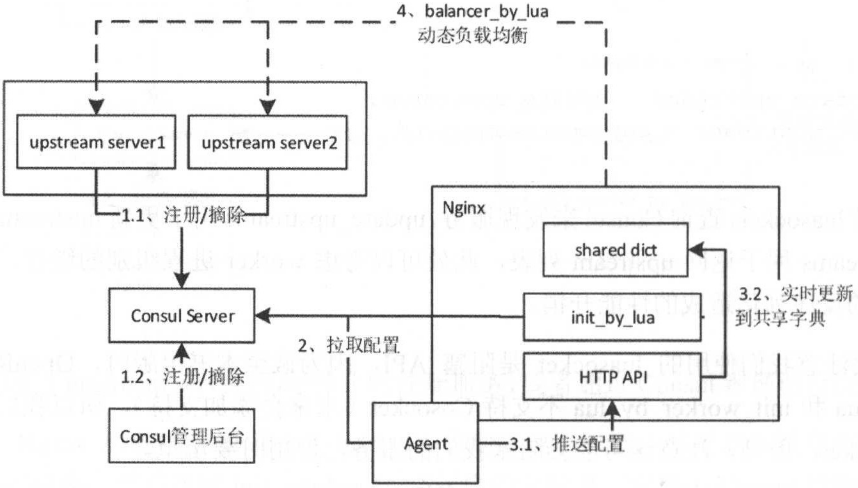
通过 luasockets 查询 Consul 来发现服务，update_upstreams 用于更新 upstream 列表，get_upstreams 用于返回 upstream 列表，此处可以考虑 worker 进程级别的缓存，减少因为 json 的反序列化造成的性能开销。

还要注意我们使用的 luasocket 是阻塞 API，因为截至本书出版时，OpenResty 在 init_by_lua 和 init_worker_by_lua 不支持 Cosocket（未来会添加支持），所以我们只能使用 luasocket，但是，注意这可能会阻塞我们的服务，使用时要慎重。

init_*_by_lua 配置

```
#存储 upstream 列表的共享字典
lua_shared_dict upstream_list 10m;

#Nginx Master 进程加载配置文件时执行，用于第一次初始化配置
init_by_lua_block {
  local dyna_upstreams = require "dyna_upstreams";
  dyna_upstreams.update_upstreams();
}
#Nginx Worker 进程调度，使用 ngx.timer.at 定时拉取配置
init_worker_by_lua_block {
  local dyna_upstreams = require "dyna_upstreams";
  local handle = nil;
  handle = function ()
    --TODO:控制每次只有一个 worker 执行
    dyna_upstreams.update_upstreams();
    ngx.timer.at(5, handle);
  end
  ngx.timer.at(5, handle);
}
```

init_worker_by_lua 是每个 Nginx Worker 进程都会执行的代码，所以实际实现时可考虑使用锁机制，保证一次只有一个人处理配置拉取。另外 ngx.timer.at 是定时轮询，不是走的长轮询，有一定的时延。有个解决方案，是在 Nginx 上暴露 HTTP API，通过主动推送的方式解决。



Agent 可以长轮询拉取，然后调用 HTTP API 推送到 Nginx 上，Agent 可以部署在 Nginx 本机或者远程。

对于拉取的配置，除了放在内存里，请考虑在本地文件系统中存储一份，在网络出问题时作为托底。

upstream 配置

```
upstream item_jd_tomcat {
  server 0.0.0.1; #占位 server
  balancer_by_lua_block {
    local balancer = require "ngx.balancer"
    local dyna_upstreams = require "dyna_upstreams";
    local upstreams = dyna_upstreams.get_upstreams();
    local ip_port = upstreams[math.random(1,table.getn(upstreams)) ]
    ngx.log(ngx.ERR, "current : =============",
                     math.random(1,table.getn (upstreams)))
    balancer.set_current_peer(ip_port.ip, ip_port.port)
  }
}
```

获取 upstream 列表，实现自己的负载均衡算法，通过 ngx.balancer API 进行动态设

置本次 upstream server。通过 balancer_by_lua 除可以实现动态负载均衡外，还可以实现个性化负载均衡算法。

最后，记得使用 lua-resty-upstream-healthcheck 模块进行健康检查。

2.9 Nginx 四层负载均衡

Nginx 1.9.0 版本起支持四层负载均衡，从而使得 Nginx 变得更加强大。目前，四层软件负载均衡器用得比较多的是 HaProxy；而 Nginx 也支持四层负载均衡，一般场景我们使用 Nginx 一站式解决方案就够了。本部分将以 TCP 四层负载均衡进行示例讲解。

2.9.1 静态负载均衡

在默认情况下，ngx_stream_core_module 是没有启用的，需要在安装 Nginx 时，添加--with-stream 配置参数启用。

```
./configure --prefix=/usr/servers --with-stream
```

1. stream 指令

我们配置 HTTP 负载均衡时，都是配置在 http 指令下，而四层负载均衡是配置在 stream 指令下。

```
stream {
    upstream mysql_backend {
        ......
    }
    server {
        ......
    }
}
```

2. upstream 配置

类似于 http upstream 配置，配置如下。

```
    upstream mysql_backend {
```

```
        server 192.168.0.10:3306 max_fails=2 fail_timeout=10s weight=1;
        server 192.168.0.11:3306 max_fails=2 fail_timeout=10s weight=1;
        least_conn;
    }
```

进行失败重试、惰性健康检查、负载均衡算法相关配置，与 HTTP 负载均衡配置类似，不再重复解释。此处我们配置实现了两个数据库服务器的 TCP 负载均衡。

3. server 配置

```
server {
    #监听端口
    listen 3308;
    #失败重试
    proxy_next_upstream on;
    proxy_next_upstream_timeout 0;
    proxy_next_upstream_tries 0;
    #超时配置
    proxy_connect_timeout 1s;
    proxy_timeout 1m;
    #限速配置
    proxy_upload_rate 0;
    proxy_download_rate 0;
    #上游服务器
    proxy_pass mysql_backend;
}
```

listen 指令指定监听的端口，默认 TCP 协议，如果需要 UDP ，则可以配置“listen 3308 udp;”。

proxy_next_upstream* 与之前讲过的 HTTP 负载均衡类似，不再重复解释。proxy_connect_timeout 配置与上游服务器连接超时时间，默认 60s。proxy_timeout 配置与客户端或上游服务器连接的两次成功读/写操作的超时时间，如果超时，将自动断开连接，即连接存活时间，通过它可以释放那些不活跃的连接，默认 10 分钟。proxy_upload_rate 和 proxy_download_rate 分别配置从客户端读数据和从上游服务器读数据的速率，单位为每秒字节数，默认为 0，不限速。

接下来就可以连接 Nginx 的 3308 端口，访问我们的数据库服务器了。

目前的配置都是静态配置，像数据库连接一般都是使用长连接，如果重启 Nginx 服务器，则会看到如下 Worker 进程一直不退出。

```
Nobody 10268 …… nginx: worker process is shutting down
```

这是因为 Worker 维持的长连接一直在使用，所以无法退出，解决办法只能是杀掉该进程。

当然，一般情况下是因为需要动态添加/删除上游服务器，才需要重启 Nginx，像 HTTP 动态负载均衡那样。如果能做到动态负载均衡，则一大部分问题就解决了。

一个选择是购买 Nginxu 商业版，另一个选择是使用 nginx-stream-upsync-module，目前，OpenResty 提供的 stream-lua-nginx-module 尚未实现 balancer_by_lua 特性，因此暂时无法使用。当前开源选择可以使用 nginx-stream-upsync-module。

2.9.2 动态负载均衡

nginx-stream-upsync-module 有一个兄弟 nginx-upsync-module，其提供了 HTTP 七层动态负载均衡，动态更新上游服务器不需要 reload nginx。当前最新版本是基于 Nginx 1.9.10 开发的，因此兼容 1.9.10+版本。其提供了基于 consul 和 etcd 进行动态更新上游服务器实现。本部分基于 Nginx 1.9.10 版本和 consul 配置中心进行演示。

首先，需要下载并添加 nginx-stream-upsync-module 模块最新版本。

```
./configure --prefix=/usr/servers --with-stream --add-module=./nginx-stream-upsync-module
```

1. upstream 配置

```
upstream mysql_backend {
    server 127.0.0.1:1111; #占位 server
    upsync 127.0.0.1:8500/v1/kv/upstreams/mysql_backend upsync_timeout =6m
           upsync_interval=500ms upsync_type=consul strong_dependency=off;
    upsync_dump_path /usr/servers/nginx/conf/mysql_backend.conf;
}
```

upsync 指令指定从 consul 哪个路径拉取上游服务器配置；upsync_timeout 配置从 consul 拉取上游服务器配置的超时时间；upsync_interval 配置从 consul 拉取上游服务器配置的间隔时间；upsync_type 指定使用 consul 配置服务器；strong_dependency 配置 nginx 在启动时是否强制依赖配置服务器，如果配置为 on，则拉取配置失败时 nginx 启动同样失败。

upsync_dump_path 指定从 consul 拉取的上游服务器后持久化到的位置，这样即使 consul 服务器出问题了，本地还有一个备份。

2. 从 Consul 添加上游服务器

```
curl  -X  PUT  -d "{\"weight\":1,  \"max_fails\":2,  \"fail_timeout\":10}"
http://127.0.0.1:8500/v1/kv/ upstreams /mysql _backend/ 10.0.0.24:3306
curl  -X  PUT -d  "{\"weight\":1,  \"max_fails\":2,  \"fail_timeout\":10}"
http://127.0.0.1:8500/v1/ kv/upstreams/mysql_backend/192.168.0.11:3306
```

3. 从 Consul 删除上游服务器

```
curl  -X  DELETE  http://127.0.0.1:8500/v1/kv/upstreams/mysql_backend/
192.168.0.11:3306
```

4. upstream_show

```
server {
    listen 1234;
    upstream_show;
}
```

配置 upstream_show 指令后，可以通过 curl http://127.0.0.1:1234/upstream_show 来查看当前动态负载均衡上游服务器列表。

到此动态负载均衡就配置完成了，我们已讲解完动态添加/删除上游服务器。在实际使用时，请进行压测来评测其稳定性。在实际应用中，更多的是用 HaProxy 进行四层负载均衡，因此，还是要根据自己的场景来选择方案。

参考资料

[1] http://nginx.org/en/docs/http/ngx_http_upstream_module.html

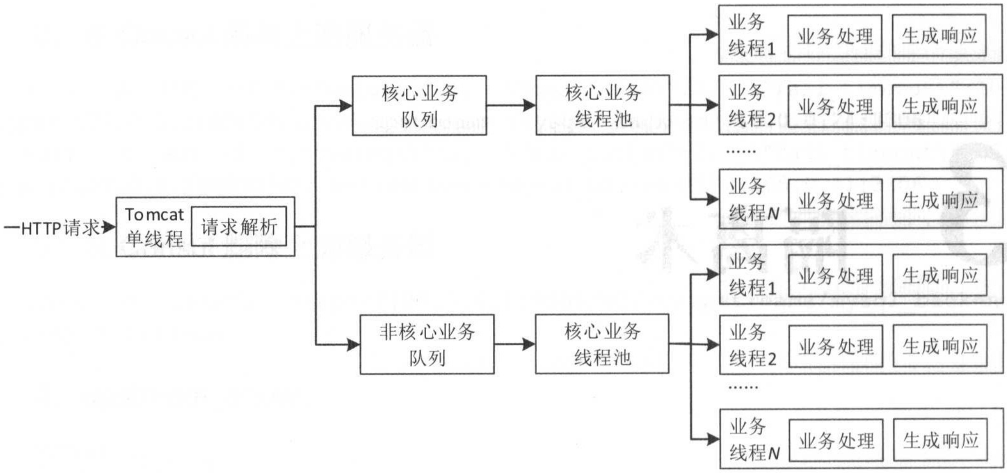
[2] http://nginx.org/en/docs/stream/ngx_stream_upstream_module.html

3 隔离术

隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时，能限定传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的。资源隔离通过隔离来减少资源竞争，保障服务间的相互不影响和可用性。笔者遇到比较多的隔离手段有线程隔离、进程隔离、集群隔离、机房隔离、读写隔离、快慢隔离、动静隔离、爬虫隔离等。出现系统问题时，可以考虑负载均衡路由、自动/手动切换分组或者降级等手段来保障可用性。

3.1 线程隔离

线程隔离主要是指线程池隔离，在实际使用时，我们会把请求分类，然后交给不同的线程池处理。当一种业务的请求处理发生问题时，不会将故障扩散到其他线程池，从而保证其他服务可用。

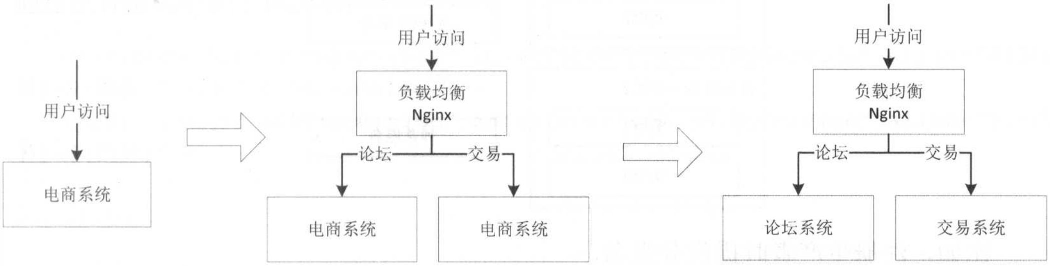


我们会根据服务等级划分两个线程池，以下是池的抽象。

```
<bean id="zeroLevelAsyncContext"
      class="com.jd.noah.base.web. DynamicAsyncContext"
      destroy-method="stop">
    <property name="asyncTimeoutInSeconds"
              value="${zero.level.request.async.timeout.seconds}"/>
    <property name="poolSize"
              value="${zero.level.request.async.pool.size}"/>
    <property name="keepAliveTimeInSeconds"
              value="${zero.level.request.async.keepalive.seconds}"/>
    <property name="queueCapacity"
              value="${zero.level.request.async.queue.capacity}"/>
</bean>
<bean id="oneLevelAsyncContext"
       class="com.jd.noah.base.web.DynamicAsyncContext"
       destroy-method="stop">
    <property name="asyncTimeoutInSeconds"
              value="${one.level.request.async.timeout.seconds}"/>
    <property name="poolSize"
              value="${one.level.request.async.pool.size}"/>
    <property name="keepAliveTimeInSeconds"
              value="${one.level.request.async.keepalive.seconds}"/>
    <property name="queueCapacity"
              value="${one.level.request.async.queue.capacity}"/>
</bean>
```

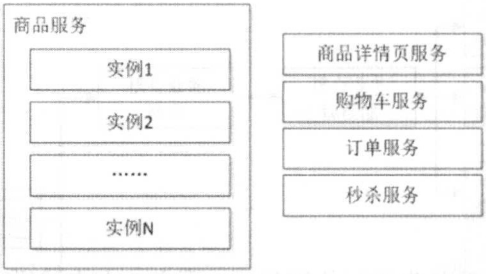
3.2 进程隔离

在公司发展初期，一般是先进行从零到一，不会一上来就进行系统拆分，这样就会开发出一些大而全的系统，系统中的一个模块/功能出现问题，整个系统就不可用了。首先，想到的解决方案是通过部署多个实例，通过负载均衡进行路由转发。但是，这种情况无法避免某个模块因 BUG 而出现如 OOM 导致整个系统不可用的风险。因此，此种方案只是一个过渡，较好的解决方案是通过将系统拆分为多个子系统来实现物理隔离。通过进程隔离使得某一个子系统出现问题时不会影响到其他子系统。

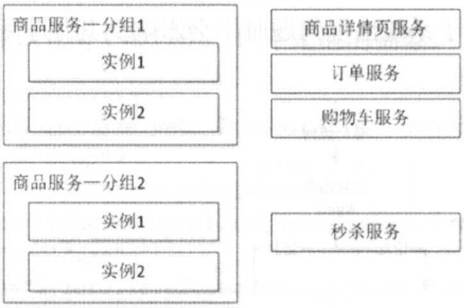


3.3 集群隔离

随着系统的发展，单实例服务无法满足需求，此时需要服务化技术，通过部署多个服务形成服务集群，来提升系统容量，如下图所示。



随着调用方的增多，当秒杀服务被刷会影响到其他服务的稳定性时，应该考虑为秒杀提供单独的服务集群，即为服务分组，这样当某一个分组出现问题时，不会影响到其他分组，从而实现了故障隔离，如下图所示。



比如，注册生产者时提供分组名。

```
<jsf:provider id="myService" interface="com.jd.MyService" alias="${分组名}" ref="myServiceImpl"/>
```

消费时使用相关的分组名即可。

```
<jsf:consumer id="myService" interface="com.jd.MyService" alias="${分组名}"/>
```

3.4 机房隔离

随着对系统可用性的要求，会进行多机房部署，每个机房的服务都有自己的服务分组，本机房的服务应该只调用本机房服务，不进行跨机房调用。其中，一个机房服务发

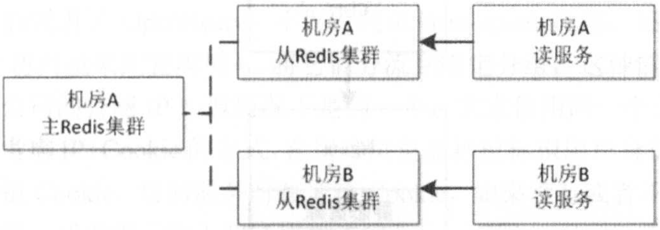
生问题时，可以通过 DNS/负载均衡将请求全部切到另一个机房，或者考虑服务能自动重试其他机房的服务，从而提升系统可用性。

一种办法是根据 IP（不同机房 IP 段不一样）自动分组，还有一种较灵活的办法是通过在分组名中加上机房名。

```
<jsf:provider id="myService" interface="com.jd.MyService" alias="${分组名}-${机房}" ref="myServiceImpl"/>
<jsf:consumer id="myService" interface="com.jd.MyService" alias="${分组名}-${机房}"/>
```

3.5 读写隔离

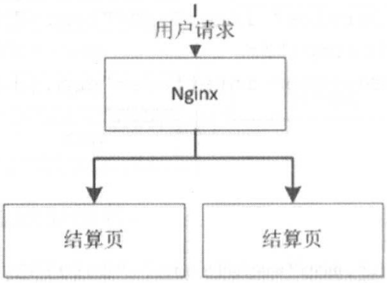
如下图所示，通过主从模式将读和写集群分离，读服务只从 Redis 集群获取数据，当主 Redis 集群出现问题时，从 Redis 集群还是可用的，从而不影响用户访问。而当从 Redis 集群出现问题时，可以进行其他集群的重试。



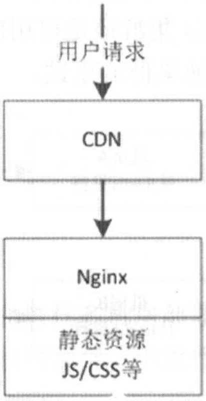
```
--先读取从
status, resp = slave_get(key)
if status == STATUS_OK then
    return status, value
end
--如果从获取失败了，从主获取
status, resp = master_get(key)
```

3.6 动静隔离

当用户访问如结算页时，如果 JS/CSS 等静态资源也在结算页系统中时，很可能因为访问量太大导致带宽被打满，从而出现不可用。

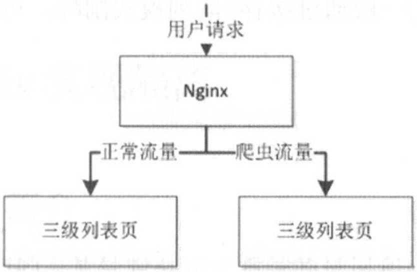


因此，应该将动态内容和静态资源分离，一般应该将静态资源放在 CDN 上，如下图所示。



3.7 爬虫隔离

在实际业务中，我们曾经统计过一些页面型应用的爬虫比例，爬虫和正常流量的比例能达到 5:1，甚至更高。而一些系统是因为爬虫访问量太大而导致服务不可用。一种解决办法是通过限流解决，另一种解决办法是在负载均衡层面将爬虫路由到单独集群，从而保证正常流量可用，爬虫流量尽量可用。



最简单的方法是使用 Nginx，可以这样配置。

```
set $flag 0;
if ($http_user_agent ~* "spider") {
   set $flag "1";
}
if($flag = "0") {
   //代理到正常集群
}
if ($flag = "1") {
   //代理到爬虫集群
}
```

实际场景我们使用了 OpenResty，不仅对爬虫 user-agent 过滤，还会过滤一些恶意 IP（通过统计 IP 访问量来配置阈值），将它们分流到固定分组，这种情况会存在一定程度的误杀，因为公司的公网 IP 一般情况下是同一个，大家使用同一个公网出口 IP 访问网站，因此，可以考虑 IP+Cookie 的方式，在用户浏览器种植标识用户身份的唯一 Cookie。访问服务前先种植 Cookie，访问服务时验证该 Cookie，如果没有或者不正确，则可以考虑分流到固定分组，或者提示输入验证码后访问。

3.8 热点隔离

秒杀、抢购属于非常合适的热点例子，对于这种热点，是能提前知道的，所以可以将秒杀和抢购做成独立系统或服务进行隔离，从而保证秒杀/抢购流程出现问题时不影响主流程。

还存在一些热点，可能是因为价格或突发事件引起的。对于读热点，笔者使用多级缓存来搞定，而写热点我们一般通过缓存+队列模式削峰，可以参考“前端交易型系统设计原则”。

3.9 资源隔离

最常见的资源，如磁盘、CPU、网络，这些宝贵的资源，都会存在竞争问题。

在“构建需求响应式亿级商品详情页”中，我们使用 JIMDB 数据同步时要 dump 数据，SSD 盘容量用了 50%以上，dump 到同一块磁盘时遇到了容量不足的问题，我们通过单独挂一块 SAS 盘来专门同步数据。还有，使用 Docker 容器时，有的容器写磁盘非常频繁，因此，要考虑为不同的容器挂载不同的磁盘。

默认 CPU 的调度策略在一些追求极致性能的场景下可能并不太适合，我们希望通过绑定 CPU 到特定进程来提升性能。当一台机器启动很多 Redis 实例时，将 CPU 通过 taskset 绑定到 Redis 实例上可以提升一些性能。还有，Nginx 提供了 worker_processes 和 worker_cpu_affinity 来绑定 CPU。如系统网络应用比较繁忙，可以考虑将网卡 IRQ 绑定到指定的 CPU 来提升系统处理中断的能力，从而提升整体性能。

可以通过 cat /proc/interrupts 查看中断情况，然后通过/proc/irq/N/smp_affinity 手动设置中断要绑定的 CPU。或者开启 irqbalance 优化中断分配，将中断均匀地分发给 CPU。

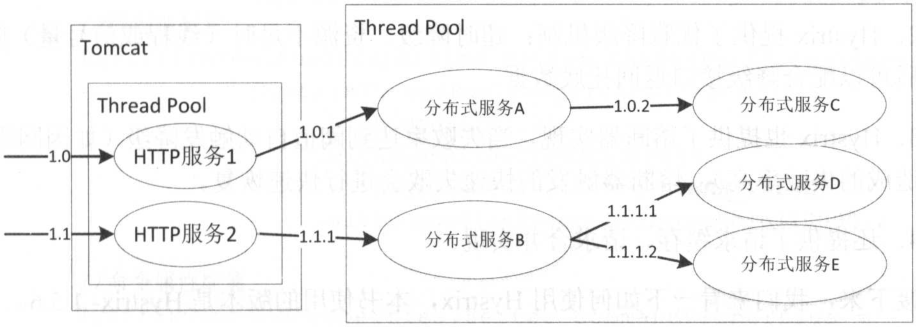
还有如大数据计算集群、数据库集群应该和应用集群隔离到不同的机架或机房，实现网络的隔离；因为大数据计算或数据库同步时会占用比较大的网络带宽，可能会拥塞网络导致应用响应变慢。

还有一些其他类似的隔离术，如环境隔离（测试环境、预发布环境/灰度环境、正式环境）、压测隔离（真实数据、压测数据隔离）、AB 测试（为不同的用户提供不同版本的服务）、缓存隔离（有些系统混用缓存，而有些系统会扔大字节值到 Redis，造成 Redis 慢查询）、查询隔离（简单、批量、复杂条件查询分别路由到不同的集群）等。通过隔离，可以将风险降到最低，将性能提升至最优。

3.10 使用 Hystrix 实现隔离

3.10.1 Hystrix 简介

Hystrix 是 Netflix 开源的一款针对分布式系统的延迟和容错库，目的是用来隔离分布式服务故障。它提供线程和信号量隔离，以减少不同服务之间资源竞争带来的相互影响；提供优雅降级机制；提供熔断机制使得服务可以快速失败，而不是一直阻塞等待服务响应，并能从中快速恢复。Hystrix 通过这些机制来阻止级联失败并保证系统弹性、可用。下图是一个典型的分布式服务实现。



首先，当大多数人在使用 Tomcat 时，多个 HTTP 服务会共享一个线程池，假设其中一个 HTTP 服务访问的数据库响应非常慢，这将造成服务响应时间延迟增加，大多数

线程阻塞等待数据响应返回，导致整个 Tomcat 线程池都被该服务占用，甚至拖垮整个 Tomcat。因此，如果我们能把不同 HTTP 服务隔离到不同的线程池，则某个 HTTP 服务的线程池满了也不会对其他服务造成灾难性故障。这就需要线程隔离或者信号量隔离来实现了。

使用线程隔离或信号隔离的目的是为不同的服务分配一定的资源，当自己的资源用完，直接返回失败而不是占用别人的资源。

同理，如“HTTP 服务 1”和“HTTP 服务 2”要分别访问远程的“分布式服务 A”和“分布式服务 B”，假设它们共享线程池，那么其中一个服务在出现问题时也会影响到另一个服务，因此，我们需要进行访问隔离，可以通过 Hystrix 的线程池隔离或信号量隔离来实现。

其次，“分布式服务 B”依赖了“分布式服务 D”和“分布式服务 E”，其中“分布式服务 D”是一个可降级的服务，意思是出现故障时（如超时、网络故障）可以暂时屏蔽掉或者返回缓存脏数据，如访问商品详情页时，可以暂时屏蔽掉上边的商家信息，不会影响用户下单流程。

当我们依赖的服务访问超时时，要提供降级策略。比如，返回托底数据阻止级联故障。当因为一些故障（如网络故障）使得服务可用率下降时，要能及时熔断，一是快速失败，二是可以保护远程分布式服务。

到此我们大体了解了 Hystrix 是用来解决什么问题的。

1. 限制调用分布式服务的资源使用，某一个调用的服务出现问题不会影响其他服务调用，通过线程池隔离和信号量隔离实现。

2. Hystrix 提供了优雅降级机制：超时降级、资源不足时（线程或信号量）降级，降级后可以配合降级接口返回托底数据。

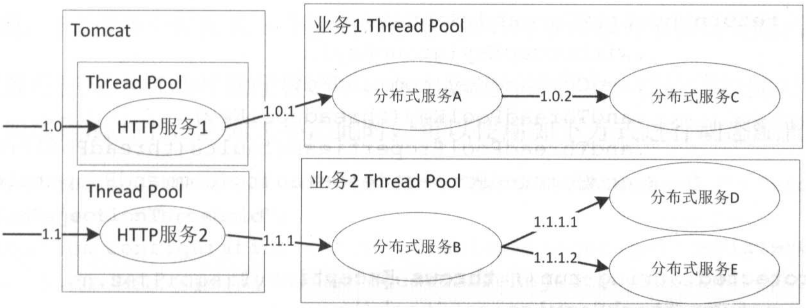
3. Hystrix 也提供了熔断器实现，当失败率达到阈值自动触发降级（如因网络故障/超时造成的失败率高），熔断器触发的快速失败会进行快速恢复。

4. 还提供了请求缓存、请求合并实现。

接下来，我们来看一下如何使用 Hystrix，本书使用的版本是 Hystrix- 1.5.6。

3.10.2 隔离示例

以线程池隔离为示例，会为不同的服务设置不同的线程池，从而实现相互隔离。



为不同的HTTP服务设置不同的线程池，为不同的分布式服务调用设置不同的线程池。

假设我们现在要调用一个获取库存服务，通过封装一个命令 GetStockServiceCommand 来实现。

```
public class GetStockServiceCommand extends HystrixCommand<String> {
    private StockService stockService;
    public GetStockServiceCommand() {
        super(setter());
    }
    private static Setter setter() {
        //服务分组
        HystrixCommandGroupKey groupKey =
                        HystrixCommandGroupKey.Factory. asKey("stock");
        //服务标识
        HystrixCommandKey commandKey =
                        HystrixCommandKey.Factory.asKey("getStock");
        //线程池名称
        HystrixThreadPoolKey threadPoolKey =
                        HystrixThreadPoolKey.Factory.asKey("stock-pool");
        //线程池配置
        HystrixThreadPoolProperties.Setter threadPoolProperties =
                    HystrixThreadPoolProperties.Setter()
                        .withCoreSize(10)
                        .withKeepAliveTimeMinutes(5)
                        .withMaxQueueSize(Integer.MAX_VALUE)
                        .withQueueSizeRejectionThreshold(10000);

        //命令属性配置
        HystrixCommandProperties.Setter commandProperties =
                    HystrixCommandProperties.Setter()
                        .withExecutionIsolationStrategy(HystrixCommandProp
erties.ExecutionIsolationStrategy.THREAD);
```

```
        return HystrixCommand.Setter
                .withGroupKey(groupKey)
                .andCommandKey(commandKey)
                .andThreadPoolKey(threadPoolKey)
                .andThreadPoolPropertiesDefaults(threadPoolProperties)
                .andCommandPropertiesDefaults(commandProperties);
    }
    @Override
    protected String run() throws Exception {
        return stockService.getStock();
    }
}
```

几个重要组件如下。

HystrixCommandGroupKey：配置全局唯一标识服务分组的名称，比如，库存系统就是一个服务分组。当我们监控时，相同分组的服务会聚合在一起，必填选项。

HystrixCommandKey：配置全局唯一标识服务的名称，比如，库存系统有一个获取库存服务，那么就可以为这个服务起一个名字来唯一识别该服务，如果不配置，则默认是简单类名。

HystrixThreadPoolKey：配置全局唯一标识线程池的名称，相同线程池名称的线程池是同一个，如果不配置，则默认是分组名，此名字也是线程池中线程名字的前缀。

HystrixThreadPoolProperties：配置线程池参数，coreSize 配置核心线程池大小和线程池最大大小，keepAliveTimeMinutes 是线程池中空闲线程生存时间（如果不进行动态配置，那么是没有任何作用的），maxQueueSize 配置线程池队列最大大小，queueSizeRejectionThreshold 限定当前队列大小，即实际队列大小由这个参数决定，通过改变 queueSizeRejectionThreshold 可以实现动态队列大小调整。

HystrixCommandProperties：配置该命令的一些参数，如 executionIsolationStrategy 配置执行隔离策略，默认是使用线程隔离，此处我们配置为 THREAD，即线程池隔离。

此处可以粗粒度实现隔离，也可以细粒度实现隔离，如下所示。

服务分组+线程池：粗粒度实现，一个服务分组/系统配置一个隔离线程池即可，不配置线程池名称或者相同分组的线程池名称配置为一样。

服务分组+服务+线程池：细粒度实现，一个服务分组中的每一个服务配置一个隔离线程池，为不同的命令实现配置不同的线程池名称即可。

混合实现：一个服务分组配置一个隔离线程池，然后对重要服务单独设置隔离线程池。

如上配置是在应用启动时就配置好了，在实际运行过程中，我们可能随时调整其中一些参数，如线程池大小、队列大小，此时，可以使用如下方式进行动态配置。

```
String dynamicQueueSizeRejectionThreshold = "hystrix.threadpool." + "stock-pool"
+ ".queueSizeRejectionThreshold";
Configuration configuration = ConfigurationManager.getConfigInstance();
configuration.setProperty(dynamicQueueSizeRejectionThreshold, 100);
```

如果是改变线程池配置，则是"hystrix.threadpool." + threadPoolKey + propertyName；如果是改变命令属性配置，则是"hystrix.command." + commandKey + propertyName。

接下来就可以通过如下方式创建命令。

```
GetStockServiceCommand command = new GetStockServiceCommand(new StockService());
```

然后通过如下方式同步调用。

```
String result = command.execute();
```

或者返回 Future 从而实现异步调用。

```
Future<String> future = command.queue();
```

或者配合 RxJava 实现响应式编程。

```
Observable<String> observe = command.observe();
observe.asObservable().subscribe((result) -> {
    System.out.println(result);
});
```

在应用 Hystrix 时，首先需要把服务封装成 HystrixCommand，即命令模式实现，然后就可以通过同步/异步/响应式模式来调用服务。

信号量隔离通过如下配置即可。

```
HystrixCommandProperties.Setter commandProperties =
    HystrixCommandProperties.Setter()
      .withExecutionIsolationStrategy(HystrixCommandProperties.Execut
ionIsolationStrategy.SEMAPHORE)
      .withExecutionIsolationSemaphoreMaxConcurrentRequests(50);
```

信号量隔离只是限制了总的并发数，服务使用主线程进行同步调用，即没有线程池。因此，如果只是想限制某个服务的总并发调用量或者调用的服务不涉及远程调用的话，可以使用轻量级的信号量来实现。

GetStockServiceCommand 不是单例，不能重用，必须每次使用创建一个。如果觉得 Hystrix 太麻烦或者太重，则可以参考 Hystrix 思路设计自己的组件。

3.11 基于 Servlet 3 实现请求隔离

在京东商品详情页系统（后端数据源）及商品详情页统一服务系统（页面中异步加载的很多服务，如库存服务、图书相关服务、延保服务等）中，我们使用了 Servlet 3 请求异步化模型，本文总结了 Servlet 3 请求异步化的一些经验。

我们将从如下几点来了解Servlet 3 异步化：为什么实现请求异步化需要使用Servlet 3、请求异步化后得到的好处是什么、如何使用 Servlet 3 异步化，以及一些 Servlet 3 异步化压测数据。

Tomcat 在收到 HTTP 请求后会按照如下流程处理请求。

1．容器负责接收并解析请求为 HttpServletRequest。

2．然后交给 Servlet 进行业务处理。

3．最后通过 HttpServletResponse 写出响应。

在 Servlet 2.x 规范中，所有这些处理都是同步进行的，也就是说必须在一个线程中完成从接收请求、业务处理到响应。

此处以 Tomcat 6 为例。Tomcat 6 没有实现 Servlet 3 规范，它在处理请求时是通过如下方式实现的。

```
org.apache.catalina.connector.CoyoteAdapter#service
        // Recycle the wrapper request and response
        if (!comet) {
            request.recycle();
            response.recycle();
        } else {
            // Clear converters so that the minimum amount of memory
            // is used by this processor
            request.clearEncoders();
            response.clearEncoders();
```

```
}
```

在请求结束时，会同步进行请求的回收，也就是说请求解析、业务处理和响应必须在一个线程内完成，不能跨越线程界限。

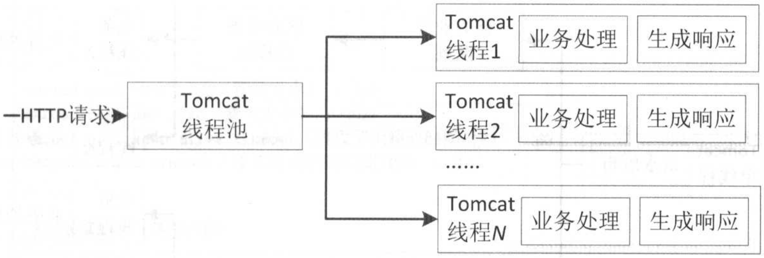
这也就说明了必须使用实现了 Servlet 3 规范的容器进行处理，如 Tomcat 7.x。

请求异步化后得到的好处如下所示。

- 基于NIO能处理更高的并发连接数，我们使用JDK 7配合Tomcat 7，压测得到不错的性能表现。
- 请求解析和业务处理线程池分离。
- 根据业务重要性对业务分级，并分级线程池。
- 对业务线程池进行监控、运维、降级等处理。

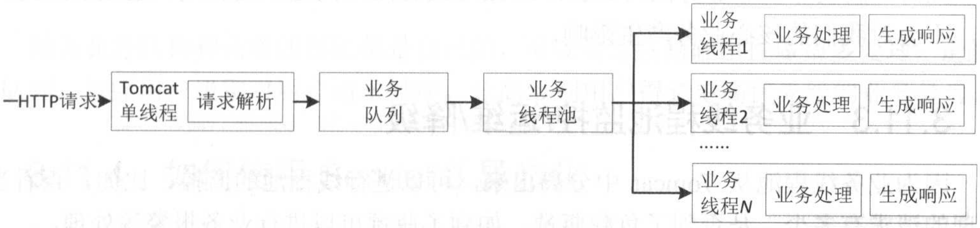
3.11.1 请求解析和业务处理线程池分离

在引入 Servlet 3 之前我们的线程模型是如下样子。



整个请求解析、业务处理、生成响应都是由 Tomcat 线程池进行处理的，而且都是在一个线程中处理，不能分离线程处理，比如接收到请求后交给其他线程处理，则不能灵活定义业务处理模型。

引入 Servlet 3 之后，我们的线程模型可以改造为如下样子。



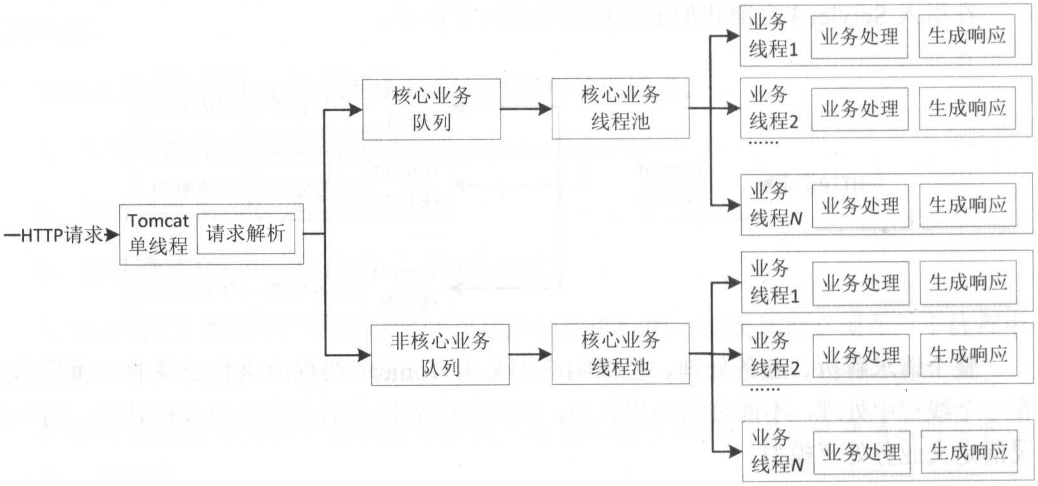
此处可以看到请求解析使用 Tomcat 单线程，而解析完成后会将请求扔到业务队列中，由业务线程池进行处理，这种处理方式可以得到如下好处。

- 根据业务重要性对业务进行分级，然后根据分级定义线程池。
- 可以拿到业务线程池，可以进行很多操作，比如监控、降级等。

3.11.2 业务线程池隔离

在一个系统的发展期间，我们一般把很多服务放到一个系统中进行处理，比如库存服务、图书相关服务、延保服务等；这些服务中，可以根据其重要性对业务分级别和做一些限制。

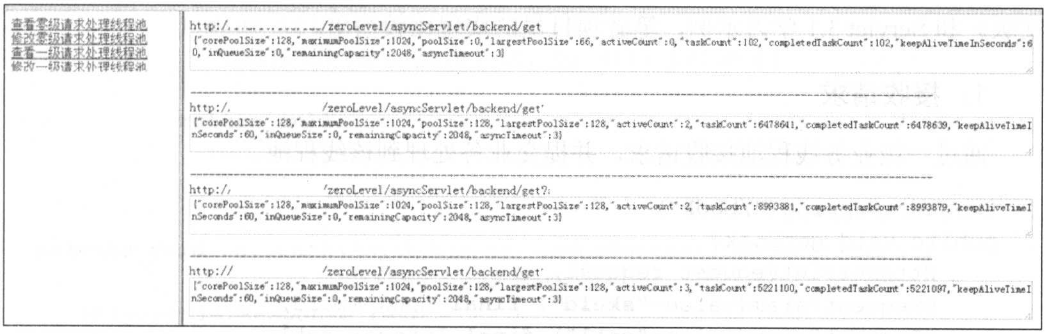
- 可以把业务分为核心业务级别和非核心业务级别。
- 为不同级别的业务定义不同的线程池，线程池之间是隔离的。
- 根据业务量定义各级别线程池大小。



此时，假设非核心业务因为数据库连接池或者网络问题引发抖动，造成响应时间过长，不会对我们的核心业务产生影响。

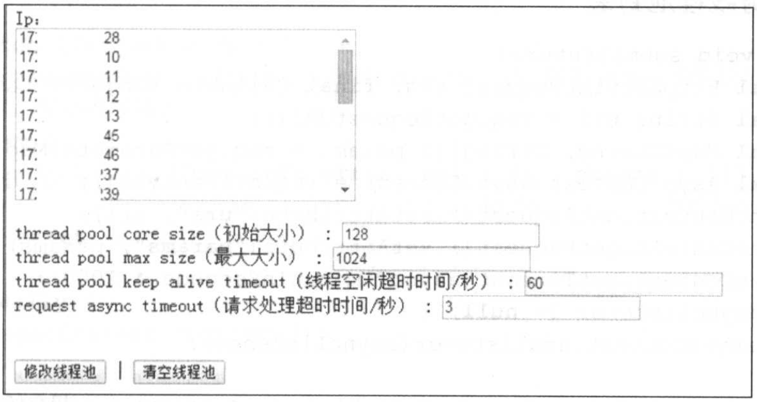
3.11.3 业务线程池监控/运维/降级

因为业务线程池从 Tomcat 中分离出来，可以进行线程池的监控，比如，查看当前处理的请求有多少，是否到了负载瓶颈，如到了瓶颈可以进行业务报警等处理。



上图是一个简陋的监控图，可实时查看到当前处理情况：正在处理的任务有多少，队列中等待的任务有多少，可以根据这些数据进行监控和预警。

另外，我们还可以进行一些简单运维。



现在，对业务线程池进行扩容，或者业务出问题时立即清空线程池防止容器崩溃等问题；而不需要进行容器重启。容器重启需要耗费数十秒甚至数几十秒，而且启动后会有预热问题，造成业务产生抖动。

如果发现请求处理不过来，负载比较高，那么最简单的办法就是直接清空线程池，将老请求拒绝掉，避免出现雪崩效应。

因为业务队列和业务线程池都是自己的，可以对这些基础组件做很多处理：定制业务队列，按照用户级别对用户请求排序，让高级别用户得到更高优先级的业务处理。

3.11.4 如何使用 Servlet 3 异步化

对于 Servlet 3 的使用，可以参考笔者博客中的文章《Servlet 3.1 规范（最终版）中

文版》和 Servlet 3.1 学习示例，笔者项目里的实现比较简单。

1. 接收请求

通过一级业务线程池接收请求，并提交业务处理到该线程池。

```
@RequestMapping("/book")
public void getBook(
    HttpServletRequest request,
    @RequestParam(value="skuId") final Long skuId,
    @RequestParam(value="cat1") final Integer cat1,
    @RequestParam(value="cat2") final Integer cat2) throws Exception {
    oneLevelAsyncContext.submitFuture(request,
                    () -> bookService.getBook (skuId, cat1, cat2));
}
```

2. 业务线程池封装

```
public void submitFuture(
    final HttpServletRequest req, final Callable <Object> task) {
    final String uri = req.getRequestURI();
    final Map<String, String[]> params = req.getParameterMap();
    final AsyncContext asyncContext = req.startAsync();  //开启异步上下文
    asyncContext.getRequest().setAttribute("uri", uri);
    asyncContext.getRequest().setAttribute("params", params);
    asyncContext.setTimeout(asyncTimeoutInSeconds * 1000);
    if(asyncListener != null) {
        asyncContext.addListener(asyncListener);
    }
    executor.submit(new CanceledCallable(asyncContext) { //提交任务给业务线程池
        @Override
        public Object call() throws Exception {
            Object o = task.call();  //业务处理调用
            if(o == null) {
                callback(asyncContext, o, uri, params);  //业务完成后，响应处理
            }
            if(o instanceof CompletableFuture) {
                CompletableFuture<Object> future =
                                (CompletableFuture <Object>)o;
                future.thenAccept(resultObject ->
                        callback(asyncContext, resultObject, uri, params))
                      .exceptionally(e -> {
                        callback(asyncContext, "", uri, params);
                        return null;
                      });
```

```
            } else if(o instanceof String) {
                callback(asyncContext, o, uri, params);
            }
            return null;
        }
    });
}
private void callback(AsyncContext asyncContext, Object result,
                      String uri, Map<String, String[]> params) {
    HttpServletResponse resp =
            (HttpServletResponse) asyncContext. getResponse();
    try {
        if(result instanceof String) {
            write(resp, (String)result);
        } else {
            write(resp, JSONUtils.toJSON(result));
        }
    } catch (Throwable e) {
        resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        // 程序内部错误
        try {
            LOG.error("get info error, uri : {}, params : {}",
                                        uri, JSONUtils.toJSON(params), e);
        } catch (Exception ex) {
        }
    } finally {
        asyncContext.complete();
    }
}
```

3. 线程池的初始化

```
@Override
public void afterPropertiesSet() throws Exception {
    String[] poolSizes = poolSize.split("-");
    //初始线程池大小
    int corePoolSize = Integer.valueOf(poolSizes[0]);
    //最大线程池大小
    int maximumPoolSize = Integer.valueOf(poolSizes[1]);
    queue = new LinkedBlockingDeque<Runnable>(queueCapacity);
    executor = new ThreadPoolExecutor(
            corePoolSize, maximumPoolSize,
            keepAliveTimeInSeconds, TimeUnit.SECONDS,
            queue);
```

```
        executor.allowCoreThreadTimeOut(true);
        executor.setRejectedExecutionHandler(
           new RejectedExecutionHandler() {
            @Override
            public void rejectedExecution(
                   Runnable r, ThreadPoolExecutor executor) {
                if(r instanceof CanceledCallable) {
                    CanceledCallable cc = ((CanceledCallable) r);
                    AsyncContext asyncContext = cc.asyncContext;
                    if(asyncContext != null) {
                        try {
                            ServletRequest req = asyncContext.getRequest();
                            String uri = (String)req.getAttribute("uri");
                            Map params = (Map)req.getAttribute("params");
                            LOG.error("async request rejected, uri : {}, params :
{}", uri, JSONUtils.toJSON(params));
                        } catch (Exception e) {}
                        try {
                            HttpServletResponse resp =
                              (HttpServletResponse) asyncContext.getResponse();
                            resp.setStatus(
                              HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
                        } finally {
                            asyncContext.complete();
                        }
                    }
                }
            }
        });

        if(asyncListener == null) {
            asyncListener = new AsyncListener() {
                @Override
                public void onComplete(AsyncEvent event) throws IOException {
                }
                @Override
                public void onTimeout(AsyncEvent event) throws IOException {
                    AsyncContext asyncContext = event.getAsyncContext();
                    try {
                        ServletRequest req = asyncContext.getRequest();
                        String uri = (String)req.getAttribute("uri");
                        Map params = (Map)req.getAttribute("params");
                        LOG.error("async request timeout, uri : {}, params : {}",
```

```
uri, JSONUtils.toJSON(params));
                } catch (Exception e) {}
                try {
                    HttpServletResponse resp =
                         (HttpServletResponse) asyncContext.getResponse();
                    resp.setStatus(
                         HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
                } finally {
                    asyncContext.complete();
                }
            }

            @Override
            public void onError(AsyncEvent event) throws IOException {
                //省略 onError 代码，与 onTimeout 类似
            }

            @Override
            public void onStartAsync(AsyncEvent event)
                                                throws IOException {
            }
        };
    }
}
```

4. 业务处理

执行 bookService.getBook(skuId, cat1, cat2)进行业务处理。

5. 返回响应

在之前封装的异步线程池上下文中直接返回。

6. Tomcat server.xml 的配置

```
<Connector port="1601" asyncTimeout="10000"
     acceptCount="10240" maxConnections="10240" acceptorThreadCount="1"
     minSpareThreads="1" maxThreads="1" redirectPort="8443"
     processorCache="1024" URIEncoding="UTF-8"
     protocol="org.apache.coyote.http11.Http11NioProtocol"
     enableLookups="false"/>
```

此处可以看到，Tomcat 线程池配置了 maxThreads=1，即一个线程进行请求解析。

3.11.5 一些 Servlet 3 异步化压测数据

压测机器基本环境：32 核 CPU、32G 内存；jdk1.7.0_71 + tomcat 7.0.57，服务响应时间在 20ms+，使用最简单的单个 URL 压测吞吐量。

1. 使用 BIO 方式压测

```
siege-3.0.7]# ./src/siege -c100 -t60s -b http://***.item.jd.com/981821
Transactions: 279187 hits
Availability: 100.00 %
Elapsed time: 59.33 secs
Data transferred: 1669.41 MB
Response time: 0.02 secs
Transaction rate: 4705.66 trans/sec
Throughput: 28.14 MB/sec
Concurrency: 99.91
Successful transactions: 279187
Failed transactions: 0
Longest transaction: 1.04
Shortest transaction: 0.00
```

2. 使用 Servlet 3 NIO 异步化压测 100 并发、60s

```
siege-3.0.7]# ./src/siege -c100 -t60s -b http://***.item.jd.com/981821
Transactions: 337998 hits
Availability: 100.00 %
Elapsed time: 59.09 secs
Data transferred: 2021.07 MB
Response time: 0.03 secs
Transaction rate: 5720.05 trans/sec
Throughput: 34.20 MB/sec
Concurrency: 149.79
Successful transactions: 337998
Failed transactions: 0
Longest transaction: 1.07
Shortest transaction: 0.00
```

3. 使用 Servlet 3 NIO 异步化压测 600 并发、60s

```
siege-3.0.7]# ./src/siege -c600 -t60s -b http://***.item.jd.com/981821
Transactions: 370985 hits
Availability: 100.00 %
Elapsed time: 59.16 secs
Data transferred: 2218.32 MB
Response time: 0.10 secs
```

```
Transaction rate: 6270.88 trans/sec
Throughput: 37.50 MB/sec
Concurrency: 598.31
Successful transactions: 370985
Failed transactions: 0
Longest transaction: 1.32
Shortest transaction: 0.00
```

可以看出，异步化之后吞吐量提升了，但响应时间也变长了。也就是说，异步化并不会提升响应时间，但会增加吞吐量和我们需要的灵活性。

通过异步化我们不会获得更快的响应时间，但是，我们获得了整体吞吐量和我们需要的灵活性：请求解析和业务处理线程池分离；根据业务重要性对业务分级，并分级线程池；对业务线程池进行监控、运维、降级等处理。

4 限流详解

在开发高并发系统时，有很多手段来保护系统，如缓存、降级和限流等。缓存目的是提升系统访问速度和增大系统处理能力，可谓是抗高并发流量的银弹。而降级是当服务出问题或者影响到核心流程的性能，需要暂时屏蔽掉，待高峰过去或者问题解决后再打开的场景。而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页）等。因此，需有一种手段来限制这些场景下的并发/请求量，这种手段就是限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。在压测时我们能找出每个系统的处理峰值，然后通过设定峰值阈值，来防止当系统过载时，通过拒绝处理过载的请求来保障系统可用。另外，也应根据系统的吞吐量、响应时间、可用率来动态调整限流阈值。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如 Nginx 的 limit_conn 模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如 Guava 的 RateLimiter、Nginx 的 limit_req 模块，用来限制每秒的平均速率），以及限制远程接口调用速率、限制 MQ 的消费速率等。另外，还可以根据网络连接数、网络流量、CPU 或内存负载等来限流。

先有缓存这个银弹，后有限流来应对 618、双 11 高并发流量，在处理高并发问题上可以说是如虎添翼，不用担心瞬间流量导致系统挂掉或雪崩，最终做到有损服务而不是

不服务。限流需要评估好，不可乱用，否则正常流量会出现一些奇怪的问题，而导致用户抱怨。

在实际应用时，也不要太纠结算法问题，因为一些限流算法实现是一样的，只是描述不一样。具体使用哪种限流技术，还是要根据实际场景来选择，不要一味去找最佳模式，白猫黑猫能解决问题的就是好猫。

因在实际工作中遇到过许多人来问如何进行限流，因此本文会详细介绍各种限流手段。接下来，我们从限流算法、应用级限流、分布式限流、接入层限流来详细学习限流技术手段。

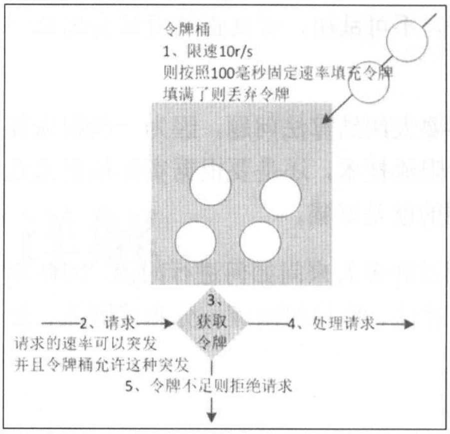
4.1 限流算法

常见的限流算法有：令牌桶、漏桶。计数器也可以用来进行粗暴限流实现。

4.1.1 令牌桶算法

令牌桶算法，是一个存放固定容量令牌的桶，按照固定速率往桶里添加令牌。令牌桶算法的描述如下。

- 假设限制2r/s，则按照500毫秒的固定速率往桶中添加令牌。
- 桶中最多存放*b*个令牌，当桶满时，新添加的令牌被丢弃或拒绝。
- 当一个*n*个字节大小的数据包到达，将从桶中删除*n*个令牌，接着数据包被发送到网络上。
- 如果桶中的令牌不足*n*个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么在缓冲区等待）。



4.1.2 漏桶算法

漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（Traffic Policing），漏桶算法的描述如下。

- 一个固定容量的漏桶，按照常量固定速率流出水滴。
- 如果桶是空的，则不需流出水滴。
- 可以以任意速率流入水滴到漏桶。
- 如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

令牌桶和漏桶算法对比如下。

- 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时，则拒绝新的请求。
- 漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝。
- 令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，或4个令牌），并允许一定程度的突发流量。
- 漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率。
- 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率。
- 两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

另外，有时我们还使用计数器来进行限流，主要用来限制总并发数，比如数据库连接池大小、线程池大小、秒杀并发数都是计数器的用法。只要全局总请求数或者一定时间段的总请求数达到设定阈值，则进行限流。这是一种简单粗暴的总数量限流，而不是平均速率限流。

到此基本的算法就介绍完了，接下来我们首先看看应用级限流。

4.2 应用级限流

4.2.1 限流总并发/连接/请求数

对于一个应用系统来说，一定会有极限并发/请求数，即总有一个 TPS/QPS 阈值，如果超了阈值，则系统就会不响应用户请求或响应得非常慢，因此我们最好进行过载保护，以防止大量请求涌入击垮系统。

如果你使用过 Tomcat，Connector 其中一种配置中有如下几个参数。

- **acceptCount**：如果Tomcat的线程都忙于响应，新来的连接会进入队列排队，如果超出排队大小，则拒绝连接；
- **maxConnections**：瞬时最大连接数，超出的会排队等待；

- **maxThreads：** Tomcat能启动用来处理请求的最大线程数，如果请求处理量一直远远大于最大线程数，则会引起响应变慢甚至会僵死。

详细的配置请参考官方文档。另外，如 MySQL（如 max_connections）、Redis（如 tcp- backlog）都会有类似的限制连接数的配置。

4.2.2 限流总资源数

如果有的资源是稀缺资源（如数据库连接、线程），而且可能有多个系统都会去使用它，那么需要加以限制。可以使用池化技术来限制总资源数，如连接池、线程池。假设分配给每个应用的数据库连接是 100，那么本应用最多可以使用 100 个资源，超出则可以等待或者抛异常。

4.2.3 限流某个接口的总并发/请求数

如果接口可能会有突发访问情况，但又担心访问量太大造成崩溃，如抢购业务，那么这个时候就需要限制这个接口的总并发/请求数总请求数了。因为粒度比较细，可以为每个接口都设置相应的阈值。可以使用 Java 中的 AtomicLong 或者 Semaphore 进行限流，在“隔离术”中也讲到了，Hystrix 在信号量模式下也使用 Semaphore 限制某个接口的总并发数。

```
try {
    if(atomic.incrementAndGet() > 限流数) {
        //拒绝请求
    }
    //处理请求
} finally {
    atomic.decrementAndGet();
}
```

这种方式适合对可降级业务或者需要过载保护的服务进行限流，如抢购业务，超出限额，要么让用户排队，要么告诉用户没货了，这对用户来说是可以接受的。而一些开放平台也会限制用户调用某个接口的试用请求量，这时就可以用这种计数器方式实现。这种方式也是简单粗暴的限流，没有平滑处理，需要根据实际情况选择使用。

4.2.4 限流某个接口的时间窗请求数

即一个时间窗口内的请求数，如想限制某个接口/服务每秒/每分钟/每天的请求数/

调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是更新量比较大有可能将基础服务打挂。这时，我们要对每秒/每分钟的调用量进行限速，一种实现方式如下所示。

```
LoadingCache<Long, AtomicLong> counter =
        CacheBuilder.newBuilder()
                .expireAfterWrite(2, TimeUnit.SECONDS)
                .build(new CacheLoader<Long, AtomicLong>() {
                    @Override
                    public AtomicLong load(Long seconds) throws Exception {
                        return new AtomicLong(0);
                    }
                });
long limit = 1000;
while(true) {
    //得到当前秒
    long currentSeconds = System.currentTimeMillis() / 1000;
    if(counter.get(currentSeconds).incrementAndGet() > limit) {
        System.out.println("限流了:" + currentSeconds);
        continue;
    }
    //业务处理
}
```

使用 Guava 的 Cache 来存储计数器，过期时间设置为 2 秒（保证能记录 1 秒内的计数）。然后，我们获取当前时间戳，取秒数来作为 key 进行计数统计和限流，这种方式简单粗暴，但应付刚才说的场景够用了。

4.2.5 平滑限流某个接口的请求数

之前的限流方式都不能很好地应对突发请求，即瞬间请求可能都被允许，从而导致一些问题。因此，在一些场景中需要对突发请求进行整形，整形为平均速率请求处理（比如 5r/s，则每隔 200 毫秒处理一个请求，平滑了速率）。这个时候有两种算法满足我们的场景：令牌桶和漏桶算法。Guava 框架提供了令牌桶算法实现，可直接拿来使用。

Guava RateLimiter 提供的令牌桶算法可用于平滑突发限流（SmoothBursty）和平滑预热限流（SmoothWarmingUp）实现。

SmoothBursty

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire());
```

```
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());

System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

将得到类似如下的输出。

```
0.0
0.198239
0.196083
0.200609
0.199599
0.19961
```

1. RateLimiter.create(5) 表示桶容量为 5 且每秒新增 5 个令牌，即每隔 200 毫秒新增一个令牌。

2. limiter.acquire()表示消费一个令牌。如果当前桶中有足够令牌，则成功（返回值为 0)，如果桶中没有令牌，则暂停一段时间。比如，发令牌间隔是 200 毫秒，则等待 200 毫秒后再去消费令牌（如上测试用例返回 0.198239，差不多等待了 200 毫秒桶中才有令牌可用)，这种实现将突发请求速率平均为固定请求速率。

再看一个突发示例。

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(5));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1));
```

将得到类似如下的输出。

0.0

0.98745

0.183553

0.199909

limiter.acquire(5)表示桶的容量为 5 且每秒新增 5 个令牌。令牌桶算法允许一定程度的突发，所以可以一次性消费 5 个令牌，但接下来的 limiter.acquire(1)将等待差不多 1 秒，桶中才能有令牌，且接下来的请求也整形为固定速率了。

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(10));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1));
```

将得到类似如下的输出。

0.0

1.997428

0.192273

0.200616

同上面的例子类似，第一秒突发了 10 个请求。令牌桶算法也允许了这种突发（允许消费未来的令牌），但接下来的 limiter.acquire(1)将等待差不多 2 秒，桶中才能有令牌，且接下来的请求也整形为固定速率了。

接下来再看一个突发的例子。

```
RateLimiter limiter = RateLimiter.create(2);
System.out.println(limiter.acquire());
Thread.sleep(2000L);
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());

System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

将得到类似如下的输出。

0.0

0.0

0.0

0.0

0.499876

0.495799

1. 创建了一个桶，容量为 2，且每秒新增 2 个令牌。

2．调用 limiter.acquire()消费一个令牌，此时令牌桶可以满足（返回值为 0）。

3．线程暂停 2 秒，接下来的两个 limiter.acquire()都能消费到令牌，第三个 limiter.acquire()也同样消费到了令牌，到第四个时就需要等待 500 毫秒了。

此处可以看到我们设置的桶容量为 2（即允许的突发量），这是因为 SmoothBursty 中有一个参数：最大突发秒数（maxBurstSeconds），默认值为 1s。突发量/桶容量=速率*maxBurstSeconds，所以本示例桶容量/突发量为 2。例子中前两个是消费了之前积攒的突发量，而第三个开始就是正常计算了。令牌桶算法允许将一段时间内没有消费的令牌暂存到令牌桶中，保留待未来使用，并允许未来请求的这种突发。

SmoothBursty 通过平均速率和最后一次新增令牌的时间计算出下次新增令牌的时间。另外，需要一个桶暂存一段时间内没有使用的令牌（即可以突发的令牌数）。另外，RateLimiter 还提供了 tryAcquire 方法来进行无阻塞或可超时的令牌消费。

因为 SmoothBursty 允许一定程度的突发，会有人担心如果允许这种突发，假设突然间来了很大的流量，那么系统很可能扛不住这种突发。因此，需要一种平滑速率的限流工具，从而在系统冷启动后慢慢趋于平均固定速率（即刚开始速率小一些，然后慢慢趋于我们设置的固定速率）。Guava 也提供了 SmoothWarmingUp 来实现这种需求，其可以认为是漏桶算法，但是在某些特殊场景又不太一样。

SmoothWarmingUp

创建方式：RateLimiter.create(doublepermitsPerSecond, long warmupPeriod, TimeUnit unit)。

permitsPerSecond 表示每秒新增的令牌数，warmupPeriod 表示从冷启动速率过渡到平均速率的时间间隔。

示例如下。

```
RateLimiter limiter = RateLimiter.create(5, 1000, TimeUnit.MILLISECONDS);
for(int i = 1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
Thread.sleep(1000L);
for(int i = 1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
```

将得到类似如下的输出。

0.0

0.51767

0.357814

0.219992

0.199984

0.0

0.360826

0.220166

0.199723

0.199555

速率是梯形上升速率，也就是说冷启动时会以一个比较大的速率慢慢达到平均速率。然后趋于平均速率（梯形下降到平均速率）。可以通过调节 warmupPeriod 参数实现一开始就是平滑固定速率。

到此应用级限流的一些方法就介绍完了。假设将应用部署到多台机器上，应用级限流方式只是单应用内的请求限流，不能进行全局限流。因此，我们需要用分布式限流和接入层限流来解决这个问题。

4.3 分布式限流

分布式限流最关键的是要将限流服务做成原子化，而解决方案可以使用 Redis+Lua 或者 Nginx+Lua 技术进行实现，通过这两种技术可以实现高并发和高性能。

首先，我们来使用 Redis+Lua 实现时间窗内某个接口的请求数限流，实现了该功能后可以改造为限流总并发/请求数和限制总资源数。Lua 本身就是一种编程语言，也可以使用它实现复杂的令牌桶或漏桶算法。

4.3.1 Redis+Lua 实现

```
local key = KEYS[1] --限流 KEY（一秒一个）
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call("INCRBY", key, "1")) --请求数+1
if current > limit then --如果超出限流大小
   return 0
elseif current == 1 then --只有第一次访问需要设置 2 秒的过期时间
   redis.call("expire", key,"2")
end
return 1
```

如上操作因是在一个 Lua 脚本中，又因 Redis 是单线程模型，因此线程安全。如上方式有一个缺点就是当达到限流大小后还是会递增的，可以改造成如下方式实现。

```
local key = KEYS[1] --限流 KEY（一秒一个）
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
   return 0
else --请求数+1，并设置 2 秒过期
   redis.call("INCRBY", key,"1")
   redis.call("expire", key,"2")
   return 1
end
```

如下是 Java 中判断是否需要限流的代码。

```
public static boolean acquire() throws Exception {
String luaScript = Files.toString(new File("limit.lua"),
Charset. defaultCharset());
Jedis jedis = new Jedis("192.168.147.52", 6379);
 //此处将当前时间戳取秒数
 String key = "ip:" + System.currentTimeMillis()/ 1000;
 Stringlimit = "3"; //限流大小
return (Long)jedis.eval(luaScript,Lists.newArrayList(key),
Lists. newArrayList(limit))==1;
}
```

因为 Redis 的限制（Lua 中有写操作，不能使用带随机性质的读操作，如 TIME），不能在 Redis Lua 中使用 TIME 获取时间戳。因此，只好从应用获取后传入，在某些极端情况下（机器时钟不准），限流会存在一些小问题。

4.3.2 Nginx+Lua 实现

```
local locks = require "resty.lock"
local function acquire()
    local lock =locks:new("locks")
    local elapsed, err =lock:lock("limit_key") --互斥锁
    local limit_counter =ngx.shared.limit_counter --计数器
    local key = "ip:" ..os.time()
    local limit = 5 --限流大小
    local current =limit_counter:get(key)

    if current ~= nil and current + 1> limit then --如果超出限流大小
        lock:unlock()
        return 0
    end
    if current == nil then
        limit_counter:set(key, 1, 1) --第一次需要设置过期时间，设置 key 的值为 1，
过期时间为 1 秒
    else
        limit_counter:incr(key, 1) --第二次开始加 1 即可
    end
    lock:unlock()
    return 1
end
ngx.print(acquire())
```

实现中我们需要使用 lua-resty-lock 互斥锁模块来解决原子性问题（在实际工程中使用时请考虑获取锁的超时问题），并使用 ngx.shared.DICT 共享字典来实现计数器。如果需要限流，则返回 0，否则返回 1。使用时需要先定义两个共享字典（分别用来存放锁和计数器数据）。

```
http {
    ......
    lua_shared_dict locks 10m;
    lua_shared_dict limit_counter 10m;
}
```

有人会纠结，如果应用并发量非常大，那么 Redis 或者 Nginx 是否能扛得住？不过，这个问题要从多方面考虑：你的流量是不是真的有这么大，是不是可以通过一致性哈希将分布式限流进行分片？是不是可以当并发量太大时降级为应用级限流？对策非常多，可以根据实际情况调节。京东目前抢购业务就是使用 Redis+Lua 来限流的，可扫二维码参考《京东抢购服务高并发实践》。

对于分布式限流，目前遇到的场景是业务上的限流，而不是流量入口的限流。流量入口限流应该在接入层完成，而接入层笔者一般使用 Nginx。

4.4 接入层限流

接入层通常指请求流量的入口，该层的主要目的有：负载均衡、非法请求过滤、请求聚合、缓存、降级、限流、A/B 测试、服务质量监控等，可以参考《使用 OpenResty 开发高性能 Web 应用》。

对于 Nginx 接入层限流可以使用 Nginx 自带的两个模块：连接数限流模块 ngx_http_limit_conn_module 和漏桶算法实现的请求限流模块 ngx_http_limit_req_module。还可以使用 OpenResty 提供的 Lua 限流模块 lua-resty-limit-traffic 应对更复杂的限流场景。

limit_conn 用来对某个 key 对应的总的网络连接数进行限流，可以按照如 IP、域名维度进行限流。limit_req 用来对某个 key 对应的请求的平均速率进行限流，有两种用法：平滑模式（delay）和允许突发模式（nodelay）。

4.4.1 ngx_http_limit_conn_module

limit_conn 是对某个 key 对应的总的网络连接数进行限流。可以按照 IP 来限制 IP 维度的总连接数，或者按照服务域名来限制某个域名的总连接数。但是，记住不是每个请求连接都会被计数器统计，只有那些被 Nginx 处理的且已经读取了整个请求头的请求连接才会被计数器统计。

1. 配置示例

```
http {
    limit_conn_zone $binary_remote_addr zone=addr:10m;
    limit_conn_log_level error;
    limit_conn_status 503;
    ...
    server {
    ...
    location /limit {
        limit_conn addr 1;
    }
```

- **limit_conn**：要配置存放key和计数器的共享内存区域和指定key的最大连接数。此处指定的最大连接数是1，表示Nginx最多同时并发处理1个连接。
- **limit_conn_zone**：用来配置限流key及存放key对应信息的共享内存区域大小。此处的key是“$binary_remote_addr”，表示IP地址，也可以使用$server_name作为key来限制域名级别的最大连接数。
- **limit_conn_status**：配置被限流后返回的状态码，默认返回503。
- **limit_conn_log_level**：配置记录被限流后的日志级别，默认error级别。

2. limit_conn 的主要执行过程

- 请求进入后首先判断当前limit_conn_zone中相应key的连接数是否超出了配置的最大连接数。
- 如果超过了配置的最大大小，则被限流，返回limit_conn_status定义的错误状态码。否则相应key的连接数加1，并注册请求处理完成的回调函数。
- 进行请求处理。
- 在结束请求阶段会调用注册的回调函数对相应key的连接数减1。

limt_conn 可以限流某个 key 的总并发/请求数，key 可以根据需要变化。

3. 按照 IP 限制并发连接数配置示例

首先，定义 IP 维度的限流区域。

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
```

接着在要限流的 location 中添加限流逻辑。

```
location /limit {
    limit_conn perip 2;
    echo "123";
```

```
}
```

即允许每个 IP 最大并发连接数为 2。

使用 AB 测试工具进行测试，并发数为 5 个，总的请求数为 5 个。

```
ab -n 5 -c 5 http://localhost/limit
```

将得到如下 access.log 输出。

```
[08/Jun/2016:20:10:51+0800] [1465373451.802] 200
[08/Jun/2016:20:10:51+0800] [1465373451.803] 200
[08/Jun/2016:20:10:51 +0800][1465373451.803] 503
[08/Jun/2016:20:10:51 +0800][1465373451.803] 503
[08/Jun/2016:20:10:51 +0800][1465373451.803] 503
```

此处我们把 access log 格式设置为 log_format main '[$time_local] [$msec] $status'; 参数分别表示时间、时间/毫秒值和响应状态码。

如果被限流，则在 error.log 中会看到类似如下的内容。

```
2016/06/08 20:10:51 [error] 5662#0: *5limiting connections by zone "perip",
client: 127.0.0.1, server: _,request: "GET /limit HTTP/1.0", host: "localhost"
```

4. 按照域名限制并发连接数配置示例

首先，定义域名维度的限流区域。

```
limit_conn_zone $server_name zone=perserver:10m;
```

接着在要限流的 location 中添加限流逻辑。

```
location /limit {
    limit_conn perserver 2;
    echo "123";
}
```

即允许每个域名最大并发请求连接数为 2。这样配置可以实现服务器最大连接数限制。

4.4.2 ngx_http_limit_req_module

limit_req 是漏桶算法实现，用于对指定 key 对应的请求进行限流，比如，按照 IP 维度限制请求速率。配置示例如下。

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

```
limit_conn_log_level error;
limit_conn_status 503;
...
server {
...
location /limit {
    limit_req zone=one burst=5 nodelay;
}
```

- **limit_req**：配置限流区域、桶容量（突发容量，默认为0）、是否延迟模式（默认延迟）。
- **limit_req_zone**：配置限流key、存放key对应信息的共享内存区域大小、固定请求速率。此处指定的key是“$binary_remote_addr”，表示IP地址。固定请求速率使用rate参数配置，支持10r/s和60r/m，即每秒10个请求和每分钟60个请求。不过，最终都会转换为每秒的固定请求速率（10r/s为每100毫秒处理一个请求，60r/m为每1000毫秒处理一个请求）。
- **limit_conn_status**：配置被限流后返回的状态码，默认返回503。
- **limit_conn_log_level**：配置记录被限流后的日志级别，默认级别为error。

limit_req 的主要执行过程如下。

（1）请求进入后首先判断最后一次请求时间相对于当前时间（第一次是 0）是否需要限流，如果需要限流，则执行步骤 2，否则执行步骤 3。

（2）如果没有配置桶容量（burst），则桶容量为 0，按照固定速率处理请求。如果请求被限流，则直接返回相应的错误码（默认为 503）。

如果配置了桶容量（burst>0）及延迟模式（没有配置 nodelay）。如果桶满了，则新进入的请求被限流。如果没有满，则请求会以固定平均速率被处理（按照固定速率并根据需要延迟处理请求，延迟使用休眠实现）。

如果配置了桶容量（burst>0）及非延迟模式（配置了 nodelay），则不会按照固定速率处理请求，而是允许突发处理请求。如果桶满了，则请求被限流，直接返回相应的错误码。

（3）如果没有被限流，则正常处理请求。

（4）Nginx 会在相应时机选择一些（3 个节点）限流 key 进行过期处理，进行内存回收。

1. 场景 2.1 测试

首先，定义 IP 维度的限流区域。

```
limit_req_zone $binary_remote_addr zone=test:10m rate=500r/s;
```

限制为每秒 500 个请求，固定平均速率为 2 毫秒一个请求。

接着在要限流的 location 中添加限流逻辑。

```
location /limit {
    limit_req zone=test;
    echo "123";
}
```

即桶容量为 0（burst 默认为 0）且延迟模式。

使用 AB 测试工具进行测试，并发数为 2 个，总的请求数为 10 个。

```
ab -n 10 -c 2 http://localhost/limit
```

将得到如下 access.log 输出。

```
[08/Jun/2016:20:25:56+0800] [1465381556.410] 200
[08/Jun/2016:20:25:56 +0800][1465381556.410] 503
[08/Jun/2016:20:25:56 +0800][1465381556.411] 503
[08/Jun/2016:20:25:56+0800] [1465381556.411] 200
[08/Jun/2016:20:25:56 +0800][1465381556.412] 503
[08/Jun/2016:20:25:56 +0800][1465381556.412] 503
```

虽然，每秒允许 500 个请求，但是，因为桶容量为 0，所以流入的请求要么被处理要么被限流，无法延迟处理。另外，平均速率在 2 毫秒左右，比如 1465381556.410 和 1465381556.411 被处理了。有朋友会说固定平均速率不是 1 毫秒吗？其实这是因为实现算法没那么精准造成的。

如果被限流在 error.log 中，则会看到如下内容。

```
2016/06/08 20:25:56 [error] 6130#0: *1962limiting requests, excess: 1.000
by zone "test", client: 127.0.0.1,server: _, request: "GET /limit HTTP/1.0",
host:"localhost"
```

如果被延迟，那么在 error.log（日志级别要 INFO 级别）中会看到如下内容。

```
2016/06/10 09:05:23 [warn] 9766#0: *97021delaying request, excess: 0.368,
by zone "test", client: 127.0.0.1,server: _, request: "GET /limit HTTP/1.0",
host:"localhost"
```

2. 场景 2.2 测试

首先，定义 IP 维度的限流区域。

```
limit_req_zone $binary_remote_addr zone=test:10m rate=2r/s;
```

为了方便测试设置速率为每秒 2 个请求，即固定平均速率是 500 毫秒一个请求。

接着在要限流的 location 中添加限流逻辑。

```
location /limit {
    limit_req zone=test burst=3;
    echo "123";
}
```

固定平均速率为 500 毫秒一个请求，通容量为 3，如果桶满了，则新的请求被限流，否则可以进入桶中排队并等待（实现延迟模式）。

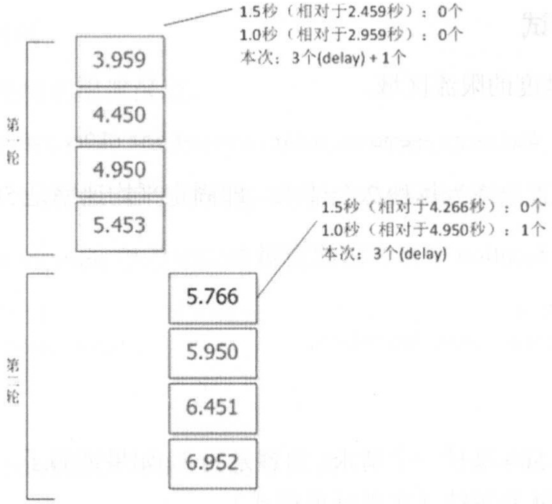
为了看出限流效果我们写了一个 req.sh 脚本。

```
ab -c 6 -n 6 http://localhost/limit
sleep 0.3
ab -c 6 -n 6 http://localhost/limit
```

首先，进行 6 个并发请求 6 次 URL，然后休眠 300 毫秒，再进行 6 个并发请求 6 次 URL。中间休眠目的是为了能跨越 2 秒看到效果，如果看不到如下的效果，则可以调节休眠时间。

将得到如下 access.log 输出。

```
[09/Jun/2016:08:46:43+0800] [1465433203.959] 200
[09/Jun/2016:08:46:43 +0800][1465433203.959] 503
[09/Jun/2016:08:46:43 +0800][1465433203.960] 503
[09/Jun/2016:08:46:44+0800] [1465433204.450] 200
[09/Jun/2016:08:46:44+0800] [1465433204.950] 200
[09/Jun/2016:08:46:45 +0800][1465433205.453] 200

[09/Jun/2016:08:46:45 +0800][1465433205.766] 503
[09/Jun/2016:08:46:45 +0800][1465433205.766] 503
[09/Jun/2016:08:46:45 +0800][1465433205.767] 503
[09/Jun/2016:08:46:45+0800] [1465433205.950] 200
[09/Jun/2016:08:46:46+0800] [1465433206.451] 200
[09/Jun/2016:08:46:46+0800] [1465433206.952] 200
```



桶容量为 3，即桶中在时间窗口内最多流入 3 个请求，且按照 2r/s 的固定速率处理请求（即每隔 500 毫秒处理一个请求）。桶计算时间窗口（1.5 秒）=速率（2r/s）/桶容量(3)，也就是说在这个时间窗口内桶最多暂存 3 个请求。因此，我们要以当前时间往前推 1.5 秒和 1 秒来计算时间窗口内的总请求数。另外，因为默认是延迟模式，所以时间窗内的请求要被暂存到桶中，并以固定平均速率处理请求。

第一轮：有 4 个请求处理成功了，按照漏桶算法桶容量应该最多 3 个才对。这是因为计算算法的问题，第一次计算因没有参考值，所以第一次计算后，后续的计算才能有参考值，因此，第一次成功可以忽略。这个问题影响很小，可以忽略。而且，可按照固定 500 毫秒的速率处理请求。

第二轮：因为第一轮请求是突发的，差不多都在 1465433203.959 时间点，只是因为漏桶将速率进行了平滑，变成了固定平均速率（每 500 毫秒一个请求）。第二轮计算时间应基于 1465433203.959。而第二轮突发请求差不多都在 1465433205.766 时间点，因此，计算桶容量的时间窗口应基于 1465433203.959 和 1465433205.766 来计算，计算结果为 1465433205.766 这个时间点漏桶为空了，可以流入桶中 3 个请求，其他请求被拒绝。又因为第一轮最后一次处理时间是 1465433205.453，所以第二轮第一个请求被延迟到了 1465433205.950。这里也要注意固定平均速率只是在配置的速率左右，存在计算精度问题，会有一些偏差。

如果桶容量改为 1（burst=1），则执行 req.sh 脚本可以看到如下输出。

```
09/Jun/2016:09:04:30+0800] [1465434270.362] 200
[09/Jun/2016:09:04:30 +0800][1465434270.371] 503
```

```
[09/Jun/2016:09:04:30 +0800] [1465434270.372]503
[09/Jun/2016:09:04:30 +0800][1465434270.372] 503
[09/Jun/2016:09:04:30 +0800][1465434270.372] 503
[09/Jun/2016:09:04:30+0800] [1465434270.864] 200

[09/Jun/2016:09:04:31 +0800][1465434271.178] 503
[09/Jun/2016:09:04:31 +0800][1465434271.178] 503
[09/Jun/2016:09:04:31 +0800][1465434271.178] 503
[09/Jun/2016:09:04:31 +0800][1465434271.178] 503
[09/Jun/2016:09:04:31 +0800][1465434271.179] 503
[09/Jun/2016:09:04:31+0800] [1465434271.366] 200
```

桶容量为 1，按照每 1000 毫秒一个请求的固定平均速率处理请求。

3. 场景 2.3 测试

首先，定义 IP 维度的限流区域。

```
limit_req_zone $binary_remote_addr zone=test:10m rate=2r/s;
```

为了方便测试配置为每秒 2 个请求，固定平均速率是 500 毫秒一个请求。

接着在要限流的 location 中添加限流逻辑。

```
location /limit {
    limit_req zone=test burst=3 nodelay;
    echo "123";
}
```

桶容量为 3，如果桶满了，则直接拒绝新请求，且每 2 秒最多两个请求，桶按照固定 500 毫秒的速率以 nodelay 模式处理请求。

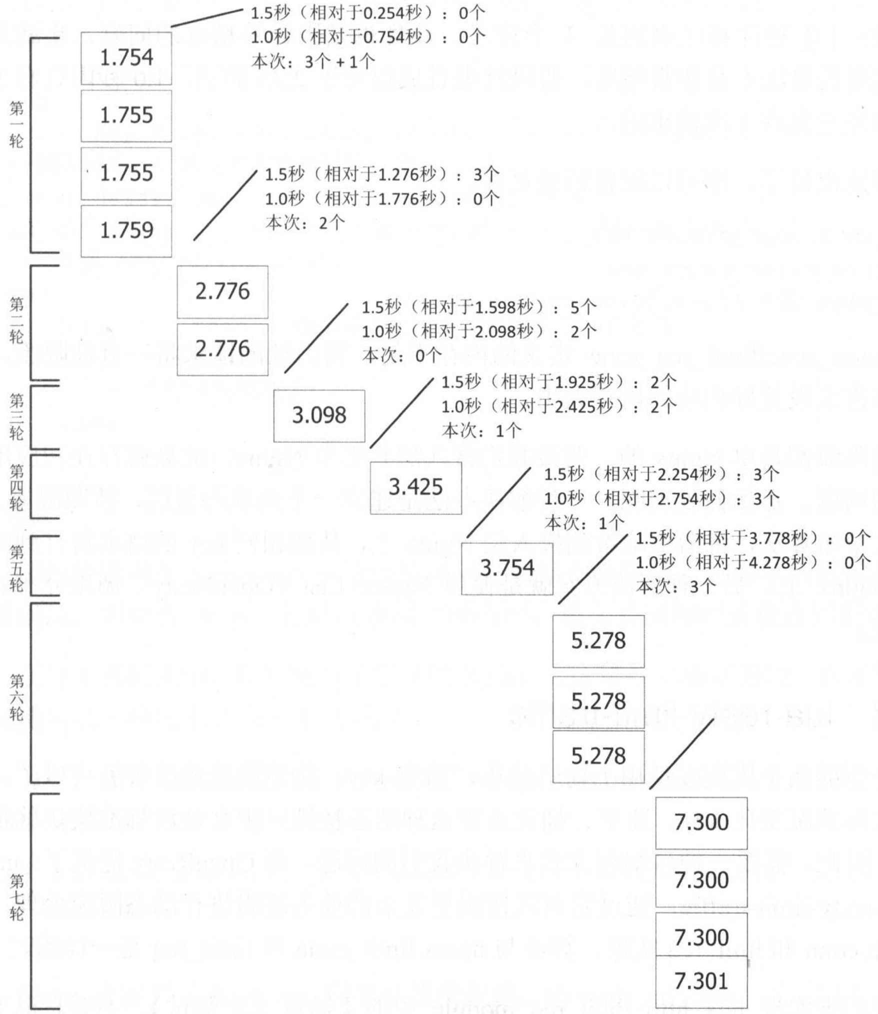
为了看到限流效果，我们写了一个 req.sh 脚本。

```
ab -c 6 -n 6 http://localhost/limit
sleep 1
ab -c 6 -n 6 http://localhost/limit
sleep 0.3
ab -c 6 -n 6 http://localhost/limit
sleep 0.3
ab -c 6 -n 6 http://localhost/limit
sleep 0.3
ab -c 6 -n 6 http://localhost/limit
sleep 2
ab -c 6 -n 6 http://localhost/limit
```

将得到类似如下的 access.log 输出。

```
[09/Jun/2016:14:30:11+0800] [1465453811.754] 200
[09/Jun/2016:14:30:11+0800] [1465453811.755] 200
[09/Jun/2016:14:30:11+0800] [1465453811.755] 200
[09/Jun/2016:14:30:11+0800] [1465453811.759] 200
[09/Jun/2016:14:30:11 +0800][1465453811.759] 503
[09/Jun/2016:14:30:11 +0800][1465453811.759] 503

[09/Jun/2016:14:30:12+0800] [1465453812.776] 200
[09/Jun/2016:14:30:12+0800] [1465453812.776] 200
[09/Jun/2016:14:30:12 +0800][1465453812.776] 503
[09/Jun/2016:14:30:12 +0800][1465453812.777] 503
[09/Jun/2016:14:30:12 +0800][1465453812.777] 503
[09/Jun/2016:14:30:12 +0800][1465453812.777] 503

[09/Jun/2016:14:30:13 +0800] [1465453813.095]503
[09/Jun/2016:14:30:13 +0800][1465453813.097] 503
[09/Jun/2016:14:30:13 +0800][1465453813.097] 503
[09/Jun/2016:14:30:13 +0800][1465453813.097] 503
[09/Jun/2016:14:30:13 +0800][1465453813.097] 503
[09/Jun/2016:14:30:13 +0800][1465453813.098] 503

[09/Jun/2016:14:30:13+0800] [1465453813.425] 200
[09/Jun/2016:14:30:13 +0800][1465453813.425] 503
[09/Jun/2016:14:30:13 +0800][1465453813.425] 503
[09/Jun/2016:14:30:13 +0800][1465453813.426] 503
[09/Jun/2016:14:30:13 +0800][1465453813.426] 503
[09/Jun/2016:14:30:13 +0800][1465453813.426] 503

[09/Jun/2016:14:30:13+0800] [1465453813.754] 200
[09/Jun/2016:14:30:13 +0800][1465453813.755] 503
[09/Jun/2016:14:30:13 +0800][1465453813.755] 503
[09/Jun/2016:14:30:13 +0800][1465453813.756] 503
[09/Jun/2016:14:30:13 +0800][1465453813.756] 503
[09/Jun/2016:14:30:13 +0800][1465453813.756] 503

[09/Jun/2016:14:30:15+0800] [1465453815.278] 200
[09/Jun/2016:14:30:15+0800] [1465453815.278] 200
[09/Jun/2016:14:30:15+0800] [1465453815.278] 200
[09/Jun/2016:14:30:15 +0800][1465453815.278] 503
[09/Jun/2016:14:30:15 +0800][1465453815.279] 503
[09/Jun/2016:14:30:15 +0800][1465453815.279] 503
```

```
[09/Jun/2016:14:30:17+0800] [1465453817.300] 200
[09/Jun/2016:14:30:17+0800] [1465453817.300] 200
[09/Jun/2016:14:30:17+0800] [1465453817.300] 200
[09/Jun/2016:14:30:17+0800] [1465453817.301] 200
[09/Jun/2016:14:30:17 +0800][1465453817.301] 503
[09/Jun/2016:14:30:17 +0800][1465453817.301] 503
```



桶容量为 3（即桶中在时间窗口内最多流入 3 个请求），且按照 2r/s 的固定速率处理请求（即每隔 500 毫秒处理一个请求）。桶计算时间窗口（1.5 秒）=速率（2r/s）/桶容量（3），也就是说在这个时间窗口内桶最多暂存 3 个请求。因此，我们要以当前时间往

前推 1.5 秒和 1 秒来计算时间窗口内的总请求数。另外，因为配置了 nodelay，是非延迟模式，所以允许时间窗内的突发请求。另外，从本示例中会看出两个问题。

第一轮和第七轮：有 4 个请求处理成功了。这是因为计算算法的问题，本示例是如果 2 秒内没有请求，然后突然来了很多请求，那么第一次计算的结果将是不正确的，这个问题影响很小，可以忽略。

第五轮：1.0 秒计算出来的是 3 个请求。此处也是因计算精度的问题，也就是说 limit_req 实现的算法不是非常精准，假设此处看成相对于 2.75 的话，1.0 秒内只有 1 次请求，所以还是允许 1 次请求的。

如果限流出错了，则可以配置错误页面。

```
proxy_intercept_errors on;
recursive_error_pages on;
error_page 503 //www.jd.com/error.aspx;
```

limit_conn_zone/limit_req_zone 定义的内存不足，则后续的请求将一直被限流，所以需要根据需求设置好相应的内存大小。

此处的限流都是单 Nginx 的，假设我们接入层有多个 Nginx，此处就存在和应用级限流相同的问题。那如何处理呢？一种解决办法是建立一个负载均衡层，按照限流 key 进行一致性哈希算法，将请求哈希到接入层 Nginx 上，从而相同 key 的请求将打到同一台接入层 Nginx 上。另一种解决方案就是使用 Nginx+Lua（OpenResty）调用分布式限流逻辑实现。

4.4.3 lua-resty-limit-traffic

之前介绍的两个模块在使用上比较简单，指定 key、指定限流速率等就可以了。如果想根据实际情况变化 key、速率、桶大小等这种动态特性，那么使用标准模块就很难去实现了。因此，需要一种可编程方式来解决我们的问题。而 OpenResty 提供了 Lua 限流模块 lua-resty-limit-traffic，通过它可以按照更复杂的业务逻辑进行动态限流处理。其提供了 limit.conn 和 limit.req 实现，算法与 nginx limit_conn 和 limit_req 是一样的。

此处我们来实现 ngx_http_limit_req_module 中的【场景 2.2 测试】，不要忘记下载 lua-resty-limit-traffic 模块并添加到 OpenResty 的 lualib 中。

配置用来存放限流用的共享字典。

```
lua_shared_dict limit_req_store 100m;
```

以下是实现【场景 2.2 测试】的限流代码 limit_req.lua。

```
local limit_req = require "resty.limit.req"
local rate = 2 --固定平均速率 2r/s
local burst = 3 --桶容量
local error_status = 503
local nodelay = false --是否需要不延迟处理
local lim, err = limit_req.new("limit_req_store", rate, burst)
if not lim then --没定义共享字典
   ngx.exit(error_status)
end
local key = ngx.var.binary_remote_addr --IP 维度的限流
--流入请求，如果请求需要被延迟，则 delay > 0
local delay, err = lim:incoming(key, true)
if not delay and err == "rejected" then --超出桶大小了
   ngx.exit(error_status)
end
if delay > 0 then --根据需要决定是延迟或者不延迟处理
   if nodelay then
      --直接突发处理了
   else
      ngx.sleep(delay) --延迟处理
   end
end
```

即限流逻辑在 Nginx access 阶段被访问，如果不被限流，则继续后续流程。如果需要被限流，则要么 sleep 一段时间继续后续流程，要么返回相应的状态码拒绝请求。

在分布式限流中，我们使用了简单的 Nginx+Lua 进行分布式限流，有了这个模块也可以使用这个模块来实现分布式限流。

另外，在使用 Nginx+Lua 时也可以获取 ngx.var.connections_active 进行过载保护，即如果当前活跃连接数超过阈值，则进行限流保护。

```
if tonumber(ngx.var.connections_active) >= tonumber(limit) then
   //限流
end
```

Nginx 也提供了 limit_rate 用来对流量限速，如 limit_rate 50k，表示限制下载速度为 50k。

到此笔者在工作中涉及的限流用法就介绍完了，这些算法中有些允许突发，有些会整形为平滑，有些计算算法简单粗暴。其中，令牌桶算法和漏桶算法实现上是类似的，

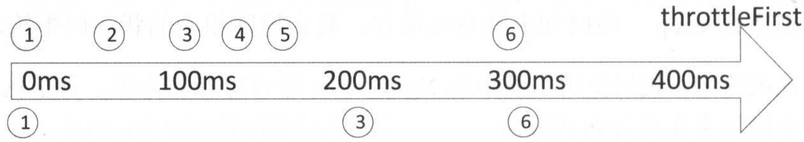
只是表述的方向不太一样，对于业务来说不必刻意去区分它们。因此，需要根据实际场景来决定如何限流，最好的算法不一定是最适用的。

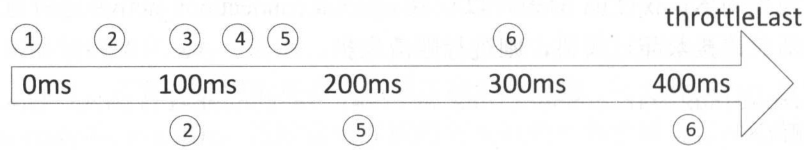
4.5 节流

有时候我们想在特定时间窗口内对重复的相同事件最多只处理一次，或者想限制多个连续相同事件最小执行时间间隔，那么可使用节流（Throttle）实现，其防止多个相同事件连续重复执行。节流主要有如下几种用法：throttleFirst、throttleLast、throttleWithTimeout。

4.5.1 throttleFirst/throttleLast

throttleFirst/ throttleLast 是指在一个时间窗口内，如果有重复的多个相同事件要处理，则只处理第一个或最后一个。其相当于一个事件频率控制器，把一段时间内重复的多个相同事件变为一个，减少事件处理频率，从而减少无用处理，提升性能。





如上图所示，throttleFirst 在一个时间窗口内只会处理该时间窗口内的第一个事件。而 throttleLast 会处理该时间窗口内的最后一个事件。

一个场景是网页中的 resize、scroll 和 mousemove 事件，当我们改变浏览器大小时会触发 resize 事件，而滚动页面元素时会触发 scroll 事件。当我们快速连续执行这些操作时会连续触发这些事件，那么可能因此造成 UI 反应慢、浏览器卡顿，因此节流就派

上用场了。对于前端开发，可以使用 jquery-throttle-debounce-plugin 实现，而 Android 开发可以使用 RxAndroid 实现。

4.5.2 throttleWithTimeout

throttleWithTimeout 也叫作 debounce（去抖），限制两个连续事件的先后执行时间不得小于某个时间窗口。

如上图所示，throttleWithTimeout 限制两个连续事件的最小间隔时间窗口。throttleFirst/ throttleLast 是基于决定时间做的处理，是以固定时间窗口为基准，对同一个固定时间窗口内的多个连续事件最多只处理一个。而 throttleWithTimeout 是基于两个连续事件的相对时间，当两个连续事件的间隔时间小于最小间隔时间窗口，就会丢弃上一个事件，而如果最后一个事件等待了最小间隔时间窗口后还没有新的事件到来，那么会处理最后一个事件。

如搜索关键词自动补全，如果用户每录入一个字就发送一次请求，而先输入的字的自动补全会被很快到来的下一个字符覆盖，那么会导致先期的自动补全是无用的。throttleWithTimeout 就是来解决这个问题的，通过它来减少频繁的网络请求，避免每输入一个字就导致一次请求。

使用 RxJava 1.2.0 实现的测试代码。

```
Observable
    .create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            //next 实现：Thread.sleep(millis); subscriber.onNext(i);
            next(subscriber, 1, 0);  //0ms
            next(subscriber, 2, 50); //50ms
            next(subscriber, 3, 50); //100ms
            next(subscriber, 4, 30); //130ms
            next(subscriber, 5, 40); //170ms
            next(subscriber, 6, 130); //300ms
            subscriber.onCompleted();
```

```
        }
    })
    .subscribeOn(Schedulers.newThread())
    .throttleWithTimeout(100, TimeUnit.MILLISECONDS)
    .subscribe(new Subscriber<Integer>() {
        ……
        @Override
        public void onNext(Integer i) {
            System.out.println("==" + i);
        }
    });
```

参考资料

[1] https://en.wikipedia.org/wiki/Token_bucket

[2] https://en.wikipedia.org/wiki/Leaky_bucket

[3] http://redis.io/commands/incr

[4] http://nginx.org/en/docs/http/ngx_http_limit_req_module.html

[5] http://nginx.org/en/docs/http/ngx_http_limit_conn_module.html

[6] https://github.com/openresty/lua-resty-limit-traffic

[7] http://nginx.org/en/docs/http/ngx_http_core_module.html#limit_rate

5 降级特技

在开发高并发系统时，有很多手段来保护系统，如缓存、降级和限流等。本章来聊聊降级策略。当访问量剧增、服务出现问题（如响应时间长或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。本文将介绍一些笔者在实际工作中遇到的或见到过的一些降级方案，供大家参考。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。降级也需要根据系统的吞吐量、响应时间、可用率等条件进行手工降级或自动降级。

5.1 降级预案

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅，从而梳理出哪些必须誓死保护，哪些可降级。比如，可以参考日志级别设置预案。

- **一般：** 比如，有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级。

- **警告**：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警。
- **错误**：比如，可用率低于90%，或者数据库连接池用完了，或者访问量突然猛增到系统能承受的最大阈值，此时，可以根据情况自动降级或者人工降级。
- **严重错误**：比如，因为特殊原因数据出现错误，此时，需要紧急人工降级。

降级按照是否自动化可分为：自动开关降级和人工开关降级。

降级按照功能可分为：读服务降级和写服务降级。

降级按照处于的系统层次可分为：多级降级。

降级的功能点主要从服务器端链路考虑，即根据用户访问的服务调用链路来梳理哪里需要降级。

- **页面降级**：在大促或者某些特殊情况下，某些页面占用了一些稀缺服务资源，在紧急情况下可以对其整个降级，以达到丢卒保帅的目的。
- **页面片段降级**：比如，商品详情页中的商家部分因为数据错误，此时，需要对其进行降级。
- **页面异步请求降级**：比如，商品详情页上有推荐信息/配送至等异步加载的请求，如果这些信息响应慢或者后端服务有问题，则可以进行降级。
- **服务功能降级**：比如，渲染商品详情页时，需要调用一些不太重要的服务（相关分类、热销榜等），而这些服务在异常情况下直接不获取，即降级即可。
- **读降级**：比如，多级缓存模式，如果后端服务有问题，则可以降级为只读缓存，这种方式适用于对读一致性要求不高的场景。
- **写降级**：比如，秒杀抢购，我们可以只进行Cache的更新，然后异步扣减库存到DB，保证最终一致性即可，此时可以将DB降级为Cache。
- **爬虫降级**：在大促活动时，可以将爬虫流量导向静态页或者返回空数据，从而保护后端稀缺资源。
- **风控降级**：如抢购/秒杀等业务，完全可以识别机器人、用户画像或者根据用户风控级别进行降级处理，直接拒绝高风险用户。

5.2 自动开关降级

自动降级是根据系统负载、资源使用情况、SLA等指标进行降级。

5.2.1 超时降级

当访问的数据库/HTTP服务/远程调用响应慢或者长时间响应慢，且该服务不是核心服务的话，可以在超时后自动降级。比如，商品详情页上有推荐内容/评价，但是，推荐内容/评价暂时不展示，对用户购物流程不会产生很大影响。对于这种服务是可以超时降级的。如果是调用别人的远程服务，则可以和对方定义一个服务响应最大时间，如果超时了，则自动降级。

注意，在实际场景中一定要配置好超时时间和超时重试次数及机制，具体细节请参考第6章。

5.2.2 统计失败次数降级

有时依赖一些不稳定的API，比如，调用外部机票服务，当失败调用次数达到一定阈值自动降级（熔断器）。然后通过异步线程去探测服务是否恢复了，恢复则取消降级。

5.2.3 故障降级

比如，要调用的远程服务挂掉了（网络故障、DNS故障、HTTP服务返回错误的状态码、RPC服务抛出异常），则可以直接降级。降级后的处理方案有：默认值（比如库存服务挂了，返回默认现货）、兜底数据（比如广告挂了，返回提前准备好的一些静态页面）、缓存（之前暂存的一些缓存数据）。

5.2.4 限流降级

当我们去秒杀或者抢购一些限购商品时，可能会因为访问量太大而导致系统崩溃，此时，开发者会使用限流来限制访问量，当达到限流阈值时，后续请求会被降级。降级后的处理方案可以是：排队页面（将用户导流到排队页面等一会儿重试）、无货（直接告知用户没货了）、错误页（如活动太火爆了，稍后重试）。

5.3 人工开关降级

在大促期间通过监控发现线上的一些服务存在问题，这个时候需要暂时将这些服务摘掉。还有，有时通过任务系统调用一些服务，但是，服务依赖的数据库可能存在：网卡打满了、数据库挂掉了或者很多慢查询，此时，需要暂停任务系统让服务方进行处理。还有发现突然调用量太大，可能需要改变处理方式（比如同步转换为异步）。此时可以使用开关来完成降级。开关可以存放到配置文件、数据库、Redis/ZooKeeper。如果不是存放在本地，则可以定期同步开关数据（比如 1 秒同步一次）。然后，通过判断某个 key 的值来决定是否降级。

另外，对于新开发的服务如果想上线进行灰度测试，但是，不太确定该服务的逻辑是否正确，此时，就需要设置开关，当新服务有问题时可以通过开关切换回老服务。还有多机房服务，如果某个机房挂掉了，则需要将一个机房的服务切到另一个机房，此时，也可以通过开关完成切换。

还有一些是因为功能问题需要暂时屏蔽掉某些功能，比如，商品规格参数数据有问题，数据问题不能用回滚解决，此时需要开关控制降级。

5.4 读服务降级

对于读服务降级一般采用的策略有：暂时切换读（降级到读缓存、降级到走静态化）、暂时屏蔽读（屏蔽读入口、屏蔽某个读服务）。在 9.4.5 节中将介绍读服务，即接入层缓存→应用层本地缓存→分布式缓存→RPC 服务/DB，我们会在接入层、应用层设置开关，当分布式缓存、RPC 服务/DB 有问题时自动降级为不调用。当然，这种情况适用于对读一致性要求不高的场景。

页面降级、页面片段降级、页面异步请求降级都是读服务降级，目的是丢卒保帅（比

如，因为这些服务也要使用核心资源，或者占了带宽影响到核心服务），或者因数据问题暂时屏蔽。

还有一种是页面静态化场景。

动态化降级为静态化：比如，平时网站可以走动态化渲染商品详情页，但是，到了大促来临之际可以将其切换为静态化来减少对核心资源的占用，而且可以提升性能。其他还有如列表页、首页、频道页都可以这么处理。可以通过一个程序定期推送静态页到缓存或者生成到磁盘，出问题时直接切过去。

静态化降级为动态化：比如，当使用静态化来实现商品详情页架构时，平时使用静态化来提供服务，但是，因为特殊原因静态化页面有问题了，需要暂时切换回动态化来保证服务正确性。

以上都保证了出问题时有预案，用户可以继续使用网站，不影响用户购物。

5.5 写服务降级

写服务在大多数场景下是不可降级的，不过，可以通过一些迂回战术来解决问题。比如，将同步操作转换为异步操作，或者限制写的量/比例。

比如，扣减库存一般这样操作。

方案 1

扣减 DB 库存，扣减成功后，更新 Redis 中的库存。

方案 2

扣减 Redis 库存，同步扣减 DB 库存，如果扣减失败，则回滚 Redis 库存。

前两种方案非常依赖 DB，假设此时 DB 性能跟不上，则扣减库存就会遇到问题。因此，我们可以想到方案 3：扣减 Redis 库存，正常同步扣减 DB 库存，性能扛不住时，降级为发送一条扣减 DB 库存的消息，然后异步进行 DB 库存扣减实现最终一致即可。

这种方式发送扣减 DB 库存消息也可能成为瓶颈，这时可以考虑方案 4：扣减 Redis

库存，正常同步扣减DB库存，性能扛不住时降级为写扣减DB库存消息到本机，然后本机通过异步进行DB库存扣减来实现最终一致性。

也就是说，正常情况下可以同步扣减库存，在性能扛不住时，降级为异步。另外，如果是秒杀场景可以直接降级为异步，从而保护系统。还有，如下单操作可以在大促时暂时降级，将下单数据写入Redis，然后等峰值过去了再同步回DB，当然也有更好的解决方案，但是更复杂，不是本书的重点。

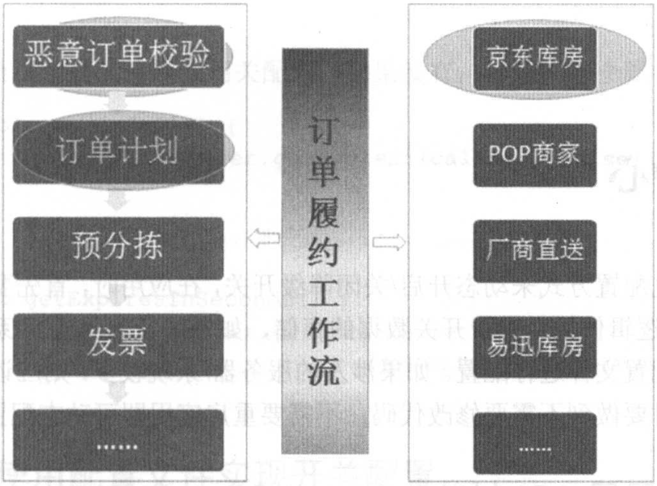
还有如用户评价，如果评价量太大，那么也可以把评价从同步写降级为异步写。当然也可以对评价按钮进行按比例开放（比如，一些人看不到评价操作按钮）。比如，评价成功后会发一些奖励，在必要的时候降级同步到异步。

5.6 多级降级

缓存是离用户越近越高效，而降级是离用户越近越对系统保护得好。因为业务的复杂性导致越到后端QPS/TPS越低。

- **页面JS降级开关**：主要控制页面功能的降级，在页面中，通过JS脚本部署功能降级开关，在适当时机开启/关闭开关。
- **接入层降级开关**：主要控制请求入口的降级，请求进入后，会首先进入接入层，在接入层可以配置功能降级开关，可以根据实际情况进行自动/人工降级。这个可以参考第17章，尤其在后端应用服务出问题时，通过接入层降级从而给应用服务有足够的时间恢复服务。
- **应用层降级开关**：主要控制业务的降级，在应用中配置相应的功能开关，根据实际业务情况进行自动/人工降级。

在下图的订单履约工作流中，整个工作流可以进行多级降级。



1. 如果恶意订单校验出现不可用的情况，则可以降级，不再同步进行恶意校验，可以直接绕过，也可以改成异步。

2. 如果订单计划出现性能下降，但还可以处理，则在这里优先处理高优级订单、处理逻辑较简单的数据（例单品单件）。

3. 分发订单时，如果仓库负载饱和，则可以降低向京东库房的输送量，增大其他目标地的输送量。

在工作流中的每一个流程中都可以进行相应的降级：优先处理高优先级数据、只处理某些特征的数据、合理分配流量到最需要的场合。上述内容由林世洪提供。

更多降级案例可扫二维码参考肖飞在 2016 年 11 月京东技术开放日分享的《服务降级背后的技术架构设计》。

5.7 配置中心

我们需要通过配置方式来动态开启/关闭降级开关，在应用时，首先要封装一套应用层 API 方便业务逻辑使用。对于开关数据的存储，如果涉及的服务器/系统较少，则初期可以考虑使用配置文件进行配置。如果涉及的服务器/系统较多，则应该使用配置中心进行配置。实现时要做到不需要修改代码，不需要重启应用即可动态配置开关。

5.7.1 应用层 API 封装

如下是我们抽象并封装的开关 API。

```
USER(
"用户信息",
"user.not.call.backend", "是否不调用后端服务",
"user.call.backend.rate.limit", "调用后端服务的限流",
"user.redis.expire.seconds", "redis 缓存过期时间"),
```

这其中涉及几个配置。

- **user.not.call.backend：** 是否回源调用后端用户服务。如果不开启，那么只会访问缓存，不会将流量打到后端。
- **user.call.backend.rate.limit：** 调用后端服务的限流，比如配置100，即一秒只有100个请求会打到后端服务，剩余请求如果缓存没有命中时，则直接返回空数据或错误。
- **user.redis.expire.seconds：** 后端返回的用户数据在缓存中缓存多久。

通过封装后，我们可以很简单地使用这些 API。

```
if (Switches.USER.notCall()) {
   return null;
}
```

或者

```
cacheService.set(CacheKeys.getUserKey(pin), info,
                      Switches.USER. getExpiresInSeconds());
```

API 实现是从配置文件获取相关配置，如果没有，则返回一个默认值。

```
public boolean notCall() {
    return DynamicConfigurer.getBoolean(callKey, false);
}
```

或者

```
public int getExpiresInSeconds() {
    return DynamicConfigurer.getInt(expiresKey,
                                     DEFAULT_EXPIRES_IN_ SECONDS);
}
```

5.7.2 使用配置文件实现开关配置

使用 properties 文件作为配置文件，借助 JDK 7 WatchService 实现文件变更监听，实现代码如下所示。

```
static {
    try {
        filename = "application.properties";
        resource = new ClassPathResource(filename);
        //监听 filename 所在目录下的文件修改、删除事件
        watchService = FileSystems.getDefault().newWatchService();
        Paths.get(resource.getFile().getParent())
            .register(watchService,
                      StandardWatchEventKinds.ENTRY_MODIFY,
                      StandardWatchEventKinds.ENTRY_DELETE);
        properties = PropertiesLoaderUtils.loadProperties(resource);
    } catch (IOException e) {e.printStackTrace();}
    //启动一个线程监听内容变化，并重新载入配置
    Thread watchThread = new Thread(() -> {
        while (true) {
            try {
                WatchKey watchKey = watchService.take();
                for (WatchEvent<?> event : watchKey.pollEvents()) {
                    if (Objects.equal(event.context().toString(),
                                      filename)) {
                        properties =
                          PropertiesLoaderUtils.loadProperties (resource);
                        break;
                    }
                }
```

```
                watchKey.reset();
            } catch (Exception e) {e.printStackTrace();}
        }
    });
    watchThread.setDaemon(true);
    watchThread.start();

    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        try {
            watchService.close();
        } catch (IOException e) {e.printStackTrace();}
    }));

}
```

- 使用WatchService监听"application.properties"文件所在目录内容的变化，包括修改、删除事件。
- 通过后台线程实现阻塞等待内容变化事件，一旦发现有内容变更，如果是"application.properties"文件发生变更，则重新装载配置文件。
- JVM停止时记得关闭WatchService。

整体实现比较简单，然后就可以封装 properties 实现自己的开关 API 了。通过配置文件实现开关配置的方式的缺点是每次配置文件内容变更都需要将配置文件同步到服务器上，这点算是比较麻烦的，如果自动部署系统支持动态更改配置文件并同步用这种方式，那么也并不麻烦。只是如果要维护多个项目时，则需要切换多个界面来操作。

5.7.3 使用配置中心实现开关配置

使用统一配置中心，或者叫分布式配置中心，目的是实现配置开关的集中管理，要有配置后台方便开关的配置，对于一般公司来说配置中心的维护要简单，不需要投入过多的人力来做这件事情。配置中心不管是采用拉取模式还是推送模式，要考虑到连接数和网络带宽可能带来的风险和问题。目前有一些开源方案可以选择，如 ZooKeeper、Diamond、Disconf、Etcd 3、Consul。本文选择使用 Consul，其支持多数据中心、服务发现、KV 存储等特性，而且使用简单，提供了简单的 Web UI 方便管理，更多介绍可以参考 Nginx 负载均衡部分。我们借助 Consul 的 KV 存储特性来实现配置管理。

启动 Consul Server 与 HTTP API CRUD 一样即可。

```
./consul agent -server -bootstrap-expect 1 -data-dir /tmp/consul  -bind
0.0.0.0 -client 0.0.0.0  -ui-dir ./ui/
```

HTTP API CRUD

- 新增/修改

```
curl -X PUT -d 'true'  http://localhost:8500/v1/kv/item_tomcat/user.not.
call.backend
curl -X PUT -d '30'  http://localhost:8500/v1/kv/item_tomcat / user.redis.
expire.seconds
```

item_tomcat 是系统名，后边是配置名，Consul 可以通过目录层次实现多级配置。

- 查询某个开关

```
curl http://localhost:8500/v1/kv/item_tomcat/user.not.call.backend
```

- 查询某个系统的开关

```
curl http://localhost:8500/v1/kv/item_tomcat?recurse
```

通过添加 recurse 参数实现目录树递归查询，可以得到如下结果。

```
[{"LockIndex":0,"Key":"item_tomcat/user.not.call.backend","Flags":0,"V
alue":"ZmFsc2U=","CreateIndex":13009,"ModifyIndex":13192},{"LockIndex":0,
"Key":"item_tomcat/user.redis.expire.seconds","Flags":0,"Value":"MzA=","C
reateIndex":13015,"ModifyIndex":13144}]
```

- 阻塞查询某个系统的开关

```
curl  "http://192.168.61.129:8500/v1/kv/item_tomcat?t=10s&recurse=  true&index
=13192"
```

此处的 index 取列表 ModifyIndex 的最大值，当其中的修改值大于此 index，则返回数据。也可以添加“wait=10s”设置超时时间，超时后阻塞返回。

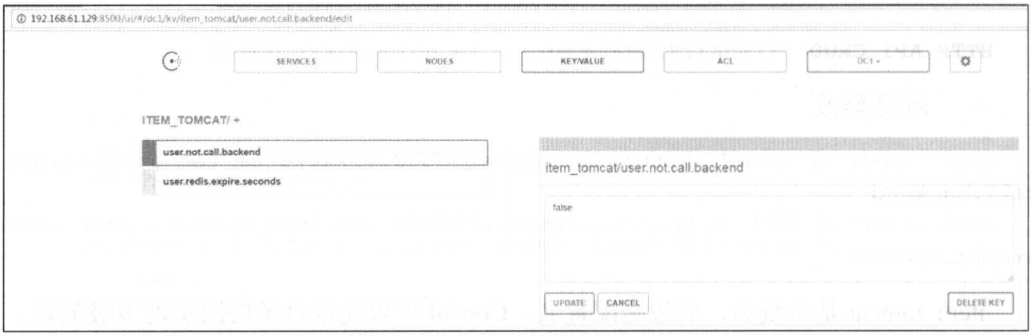
- 删除某个开关

```
curl -X DELETE http://localhost:8500/v1/kv/item_tomcat/user.not. call.
backend
```

- 删除某个系统开关

```
curl -X DELETE http://localhost:8500/v1/kv/item_tomcat?recurse
```

整体使用比较简单，Consul Web UI 提供了可视化配置，在启动时，通过 ui-dir 指定下载的 Web UI 目录即可，配置界面如下图所示。



配置界面十分简洁，目前存在的一个缺点是没有配置项的描述功能，在定义配置时，要起一个好理解且清晰的名字。

接下来就需要在应用代码中引入配置中心了，代码如下所示。

```
private static transient Properties properties = null;
private static transient String system = "item_tomcat";
static {
    Consul consul = Consul.builder()
        .withHostAndPort(HostAndPort.fromString("192.168.61.129:8500"))
        .withConnectTimeoutMillis(1000)
        .withReadTimeoutMillis(30 * 1000)
        .withWriteTimeoutMillis(5000).build();
    final KeyValueClient keyValueClient = consul.keyValueClient();
    final AtomicBoolean needBreak = new AtomicBoolean(true);
    Thread thread = new Thread(() -> {
        BigInteger index = BigInteger.ZERO;
        while(true) {
            Properties _properties = new Properties();
            try {
                //阻塞获取 item_tomcat 下的数据（阻塞 30 秒），
                //index 是 item_tomcat 下的最后修改数据的修改 index，为了实现阻塞，
                //此处阻塞时间受 readTimeoutMillis 影响
                List<Value> values = keyValueClient.getValues(system,
                             QueryOptions.blockSeconds(30, index).build());
                for(Value value : values) {
                    _properties.put(
                           value.getKey().substring(system.length() + 1),
                           value.getValueAsString().orNull());
                    //获取最大的一个最后修改 index，
                    //实现 KeyValueClient #getValues 的阻塞访问
                    index = index.max(
                                 BigInteger.valueOf(value. getModifyIndex()));
```

```
                }
                properties = _properties;
            } catch (ConsulException e) {
                e.printStackTrace();
                if(e.getCode() == 404) { //如果 key 不存在，则休眠
                    try { Thread.sleep(5000L); } catch (Exception e1) {}
                }
            }
            if(needBreak.get() == true) {break;}
        }
    });
    thread.run();//先运行一次
    needBreak.set(false);
    thread.setDaemon(true);
    thread.start();
}
```

- 在配置Consul时，目前我们使用的是IP/PORT，实际应用时建议使用域名/VIP，记得配置相关的超时时间。
- KeyValueClient在获取数据时使用拉取模式（长轮询），可以设置合理的阻塞时间（此时间受限于Consul配置的超时时间），选择最大的ModifyIndex进行阻塞等待。
- 当ConsulException的code=404表示system在配置中心没有任何配置。
- 在加载该类时先运行一次拉取配置，然后启动后台线程阻塞拉取最新配置。

Consul 的一个缺点是无法进行增量配置更新，如果订阅配置的应用很多，那么每次配置更新的下发量就非常大，如果有增量配置更新的话，则只需要把变化的进行下发。

到此集成 Consul 配置中心就完成了，此处只列出了核心代码，还有一些异常情况需要大家处理，使得配置中心在应用中做到高可用。

在第 3 章中的“使用 Hystrix 实现隔离”部分我们已经介绍了 Hystrix 的作用，也通过 Hystrix 实现了线程隔离和信号量隔离，本部分将介绍使用 Hystrix 实现降级和熔断。

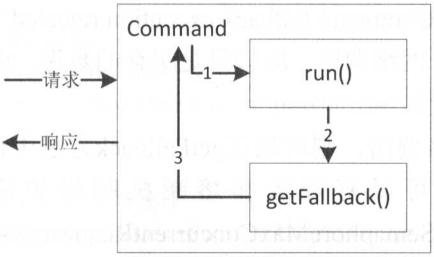
5.8 使用 Hystrix 实现降级

通过配置中心可以进行人工降级，而我们也需要根据服务的超时时间进行自动降级，本部分将演示使用 Hystrix 实现超时自动降级。Hystrix 的介绍请参考第 3 章中的 Hystrix 简介部分。

```java
public class GetStockServiceCommand extends HystrixCommand<String> {
    private StockService stockService;
    public GetStockServiceCommand(StockService stockService) {
        super(setter());
        this.stockService = stockService;
    }
    private static Setter setter() {
        //服务分组
        HystrixCommandGroupKey groupKey =
                HystrixCommandGroupKey.Factory. asKey("stock");
        ……
        //命令配置
        HystrixCommandProperties.Setter commandProperties =
              HystrixCommandProperties.Setter()
               ……
               .withExecutionIsolationStrategy(HystrixCommandProperties.
ExecutionIsolationStrategy.THREAD)
               .withFallbackEnabled(true)//默认为 true
               //默认为 10
               .withFallbackIsolationSemaphoreMaxConcurrentRequests(100)
               //默认为 false
               .withExecutionIsolationThreadInterruptOnFutureCancel(true)
                //默认为 true
               .withExecutionIsolationThreadInterruptOnTimeout(true)
               .withExecutionTimeoutEnabled(true) //默认为 true
               .withExecutionTimeoutInMilliseconds(1000)//默认为 1000
               ;

        return HystrixCommand.Setter
                      .withGroupKey(groupKey)
                      .andCommandPropertiesDefaults(commandProperties);
```

```
    }

    @Override
    protected String run() throws Exception {
        //可以通过抛出异常，或 Thread.sleep 模拟超时
        return stockService.getStock();
    }
    @Override
    protected String getFallback() {//降级方法
        return "有货";
    }
}
```

整体执行流程如下图所示。



首先，Command 会调用 run 方法，如果 run 方法超时或者抛出异常，且启用了降级处理，则调用 getFallback 方法进行降级。

而降级处理主要进行两部分处理：HystrixCommandProperties 配置和 getFallback 降级处理方法。首先，我们看一下 HystrixCommandProperties 配置。

- **withFallbackEnabled**：是否启用降级处理，如果启用了，则在超时或异常时调用getFallback进行降级处理，默认为开启。
- **withFallbackIsolationSemaphoreMaxConcurrentRequests**：fallback方法的信号量配置，配置getFallback方法并发请求的信号量，如果请求超过了并发信号量限制，则不再尝试调用getFallback方法，而是快速失败，默认信号量为10。
- **withExecutionIsolationThreadInterruptOnFutureCancel**：隔离策略为THREAD时，当执行线程执行超时时，是否进行中断处理，即Future#cancel(true)处理，默认为false。
- **withExecutionIsolationThreadInterruptOnTimeout**：当隔离策略为THREAD时，当执行线程执行超时时，是否进行中断处理，默认为true。

- **withExecutionTimeoutEnabled**：是否启用执行超时机制，默认为true。
- **withExecutionTimeoutInMilliseconds**：执行超时时间，默认为1000毫秒，如果命令是线程隔离，且配置了executionIsolationThreadInterruptOnTimeout=true，则执行线程将执行中断处理。如果命令是信号量隔离，则进行终止操作，因为信号量隔离与主线程是在一个线程中执行，其不会中断线程处理，所以要根据实际情况来决定是否采用信号量隔离，尤其涉及网络访问的情况。

当开启了降级处理，run 方法超时或者异常时将会调用 getFallback 处理，使用 getFallback 时需要注意以下几点。

- 其最大并发数受fallbackIsolationSemaphoreMaxConcurrentRequests控制，因此，如果失败率非常高，则要重新配置该参数，如果最大并发数超了该配置，则不会再执行getFallback，而是快速失败，抛出如“HystrixRuntimeException: GetStockServiceCommand fallback execution rejected”的异常。
- 该方法不能进行网络调用，应该只是缓存的数据，或者静态数据（如我们的库存方法返回有货）。
- 如果必须走网络调用，则应该在getFallback方法中调用另一个Command实现，通过Command可以有降级和熔断机制保护应用，而getFallback只有fallbackIsolationSemaphoreMaxConcurrentRequests参数控制最大并发数。

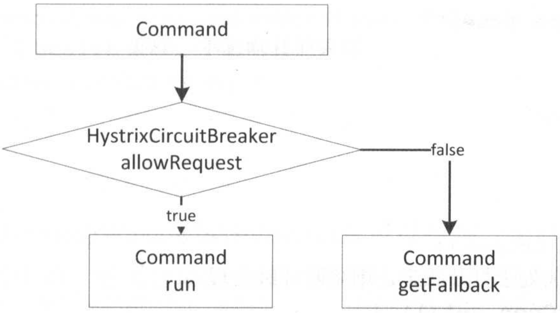
在使用 Command 的业务代码处，可以使用如下方法获取执行的状态。

- **isResponseTimedOut**：响应是否超时了。
- **isFailedExecution**：执行是否失败了，如抛出了异常。
- **getFailedExecutionException**：获取失败后的执行异常，即run方法抛出的异常。
- **isResponseFromFallback**：是否是getFallback返回的响应。

5.9 使用 Hystrix 实现熔断

5.9.1 熔断机制实现

Hystrix 提供了熔断实现，熔断后会自动降级处理，如下图所示。



Command 首先调用 HystrixCircuitBreaker#allowRequest 判断是否熔断了，如果没有熔断，则执行 Command#run 方法正常处理，如果熔断了，则直接调用 Command#getFallback 方法降级处理。

接下来，我们看一下 HystrixCircuitBreakerImpl# allowRequest 方法的实现。

```
public boolean allowRequest() {
    //如果熔断开关强制打开，则熔断降级处理
    if (properties.circuitBreakerForceOpen().get()) {
        return false;
    }
    //如果熔断开关强制闭合，则正常处理
    if (properties.circuitBreakerForceClosed().get()) {
        //还是需要调用 isOpen 方法进行采样处理
        isOpen();
        return true;
    }
    //正常判断
    return !isOpen() || allowSingleTest();
}
//允许在一个时间窗口内进行单次访问测试
public boolean allowSingleTest() {
   //熔断开关打开时，最后一次测试时间
   long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get();
   //如果熔断开关处于打开状态，
   //且在一个时间窗口内（circuitBreakerSleepWindowInMilliseconds），
   //则允许一次访问进行测试
    if (circuitOpen.get() &&
      System.currentTimeMillis() >
         timeCircuitOpenedOrWasLastTested +
         properties.circuitBreakerSleepWindowInMilliseconds().get()) {
if (circuitOpenedOrLastTestedTime.
         compareAndSet(timeCircuitOpenedOrWasLastTested,
                  System.currentTimeMillis())) {
```

```
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean isOpen() {
        //如果熔断开关处于打开状态，则熔断降级处理
        if (circuitOpen.get()) {
            return true;
        }

        //熔断开关当前处于闭合状态，需要根据采样判断当前是否需要熔断
        HealthCounts health = metrics.getHealthCounts();
        //如果当前采样的总请求数小于 circuitBreakerRequestVolumeThreshold 阀值，
        //则不进行熔断
        if (health.getTotalRequests() <
                properties.circuitBreakerRequestVolumeThreshold(). get()) {
            return false;
        }
        //如果当前采样的错误率小于 circuitBreakerErrorThresholdPercentage 阀值，
        //则不进行熔断
        //errorPercentage = errorCount / totalCount * 100
        if (health.getErrorPercentage() <
                properties. circuitBreakerErrorThresholdPercentage().get()) {
            return false;
        } else {
            //当前失败率超过了阀值，进行熔断降级处理
            if (circuitOpen.compareAndSet(false, true)) {
                circuitOpenedOrLastTestedTime
                           .set(System.currentTimeMillis());
                return true;
            } else {
                return true;
            }
        }
    }
```

当我们的熔断开关处于打开状态时，此时是不允许处理任何请求的，而是直接降级处理，但是提供了 markSuccess 方法，当请求处理成功时进行熔断开关闭合。

```
public void markSuccess() {
    if (circuitOpen.get()) {
```

```
        if (circuitOpen.compareAndSet(true, false)) {
            //重置 health 采样，不影响其他采样
            metrics.resetStream();
        }
    }
}
```

通过 circuitBreakerSleepWindowInMilliseconds 可以控制一个时间窗口内可进行一次请求测试，如果测试成功，则闭合熔断开关，否则还是打开状态，从而实现了快速失败和快速恢复。

关于熔断开关需要知道如下几个概念。

- **闭合（Closed）**：如果配置了熔断开关强制闭合，或者当前请求失败率没有超过失败率阈值，则熔断开关处于闭合状态，不启动熔断机制，即不进行降级处理。
- **打开（Open）**：如果配置了熔断开关强制打开，或者当前失败率超过失败率阈值，则熔断开关打开，启动熔断机制，根据配置调用降级处理方法getFallback进行降级处理。
- **半打开（Half-Open）**：当熔断处于打开状态后，不能一直熔断下去，需要在一个时间窗口后进行重试，这种状态就是半打开。Hystrix 允许在 circuitBreakerSleepWindowInMilliseconds窗口内进行一次重试，重试成功则闭合熔断开关，否则熔断开关还是处于打开状态。

那什么样的请求被认为是错误呢，HealthCounts 在统计错误数量时使用如下方法。

```
public HealthCounts plus(long[] eventTypeCounts) {
    long updatedTotalCount = totalCount;
    long updatedErrorCount = errorCount;

    long successCount = eventTypeCounts[HystrixEventType.SUCCESS.ordinal()];
    long failureCount = eventTypeCounts[HystrixEventType.FAILURE.ordinal()];
    long timeoutCount = eventTypeCounts[HystrixEventType.TIMEOUT.ordinal()];
    long threadPoolRejectedCount =
eventTypeCounts[HystrixEventType.THREAD_POOL_REJECTED.ordinal()];
    long semaphoreRejectedCount =
      eventTypeCounts[HystrixEventType.SEMAPHORE_REJECTED.ordinal()];

    updatedTotalCount += (successCount + failureCount + timeoutCount +
                          threadPoolRejectedCount + semaphoreRejectedCount);
    updatedErrorCount += (failureCount + timeoutCount +
                          threadPoolRejectedCount + semaphoreRejectedCount);
```

```
    return new HealthCounts(updatedTotalCount, updatedErrorCount);
}
```

即失败（如异常）、超时、线程池拒绝、信号量拒绝数量总和是失败总数。

5.9.2 配置示例

下面是 HystrixCommandProperties 的熔断参数配置。

```
HystrixCommandProperties.Setter commandProperties =
    HystrixCommandProperties. Setter()
       ……
       .withCircuitBreakerEnabled(true)//默认为true
       .withCircuitBreakerForceClosed(false)//默认为false
       .withCircuitBreakerForceOpen(false)//默认为false
       .withCircuitBreakerErrorThresholdPercentage(50)//默认为50%
       .withCircuitBreakerRequestVolumeThreshold(20) //默认为20
       .withCircuitBreakerSleepWindowInMilliseconds(5000)//默认为5s
```

具体配置含义如下所示。

- **withCircuitBreakerEnabled**：是否开启熔断机制，默认为true。
- **withCircuitBreakerForceClosed**：是否强制关闭熔断开关，如果强制关闭了熔断开关，则请求不会被降级，一些特殊场景可以动态配置该开关，默认为false。
- **withCircuitBreakerForceOpen**：是否强制打开熔断开关，如果强制打开了熔断开关，则请求强制降级调用getFallback处理，可以通过动态配置来打开该开关实现一些特殊需求，默认为false。
- **withCircuitBreakerErrorThresholdPercentage**：如果在一个采样时间窗口内，失败率超过该配置，则自动打开熔断开关实现降级处理，即快速失败。默认配置下采样周期为10s，失败率为50%。
- **withCircuitBreakerRequestVolumeThreshold**：在熔断开关闭合的情况下，在进行失败率判断之前，一个采样周期内必须进行至少*N*个请求才能进行采样统计，目的是有足够的采样使得失败率计算正确，默认为20。
- **withCircuitBreakerSleepWindowInMilliseconds**：熔断后的重试时间窗口，且在该时间窗口内只允许一次重试。即在熔断开关打开后，在该时间窗口允许有一次重试，如果重试成功，则将重置Health采样统计并闭合熔断开关实现快速恢复，否则熔断开关还是打开状态，执行快速失败。
- 熔断后将降级调用getFallback进行处理（fallbackEnabled=true），通过Command如下方法可以判断是否熔断了。

- **isCircuitBreakerOpen**：熔断开关是否打开了，通过“circuitBreakerForceOpen().get() || (!circuitBreakerForceClosed().get() && circuitBreaker.isOpen())”判断。
- **isResponseShortCircuited**：isCircuitBreakerOpen=true，且调用getFallback时返回true。

5.9.3 采样统计

Hystrix 在内存中存储采样数据，支持如下三种采样。

- **BucketedCounterStream**：计数统计，比如记录一定时间窗口内的失败、超时、线程池拒绝、信号量拒绝数量，记录*N*组。写入数据时写到第*N*组，统计时使用前*N*-1组数据，因为第*N*个刚开始统计时是随时变化的。然后基于时间滚转采样分组即可。

成功	1	2	3	10	15	1	2	3	10	15
失败	10	8	10	0	0	10	8	10	0	0
超时	1	1	0	0	0	1	1	0	0	0
线程拒绝	0	0	0	0	0	0	0	0	0	0

  采样统计滚转时间窗口为10s，每秒1个分组（桶），即每秒采样一次，每个分组记录着当前桶的成功、失败、超时、线程拒绝统计数量。

- **RollingConcurrencyStream**：最大并发数统计，如Command/ThreadPool的最大并发数。
- **RollingDistributionStream**：延时百分比统计，同HystrixRollingNumber类似，差别在于其是百分位数的统计。比如每组记录P（比如100）个数值，统计时使用前*N*-1组数据，将分组内数据按从小到大排序，然后累加，处于第*p*%位置的数值就是*p*百分位数，通过它可以实现P50、P99、P999，Hystrix用来统计时延的分布情况。最新版本Hystrix使用HdrHistogram库来实现统计。

1. Command、ThreadPool 计数/最大并发采样统计

```
HystrixThreadPoolProperties.Setter threadPoolProperties =
    HystrixThreadPoolProperties.Setter()
      ......
      .withMetricsRollingStatisticalWindowInMilliseconds(1000)
      .withMetricsRollingStatisticalWindowBuckets(10);
```

```
HystrixCommandProperties.Setter commandProperties =
     HystrixCommandProperties. Setter()
       ……
       .withMetricsRollingStatisticalWindowInMilliseconds(10000)
       .withMetricsRollingStatisticalWindowBuckets(10);
```

- **withMetricsRollingStatisticalWindowInMilliseconds**：配置采样统计滚转时间窗口，默认为10s。
- **withMetricsRollingStatisticalWindowBuckets**：配置采用统计滚转时间窗口内的桶的总数量，默认为10，比如时间窗口为10000，桶数量为10，则采样统计间隔为每秒一个桶统计。

2. Command 健康度采样统计

```
HystrixCommandProperties.Setter commandProperties =
     HystrixCommandProperties. Setter()
       ……
       .withMetricsRollingStatisticalWindowInMilliseconds(10000)
       .withMetricsHealthSnapshotIntervalInMilliseconds(500);
```

- **withMetricsRollingStatisticalWindowInMilliseconds**：同上所示。
- **withMetricsHealthSnapshotIntervalInMilliseconds**：记录健康采用统计的快照频率，默认为500ms，即500ms一个采样统计间隔，那么桶的数量为10000/500=20个。

该统计在熔断机制中使用，如果计算熔断的频率非常高，则要控制好采样的频率，如果太频繁，那么将造成 CPU 计算密集，如 10ms 一个周期，因为会对前 *N*-1 个桶进行统计，计算累加时会耗费 CPU。所以选择 Hystrix 要注意此处的性能消耗和调优。如果此处是性能瓶颈，则可以废掉统计，或者按照 Hystrix 思路实现自己的降级组件。

3. Command 时延分布采样统计

```
HystrixCommandProperties.Setter commandProperties =
     HystrixCommandProperties.Setter()
       ……
       .withMetricsRollingPercentileWindowInMilliseconds(60000)
       .withMetricsRollingPercentileWindowBuckets(6);
```

同 withMetricsRollingStatisticalWindowInMilliseconds 和 withMetricsRollingStatisticalWindowBuckets，默认采样滚转时间窗口为 60s，总共 6 个桶，即采样统计间隔为每 10 秒一个桶统计。

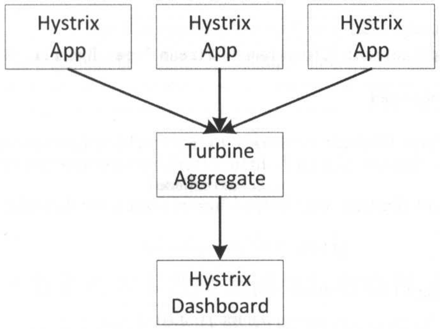
4. 统计结果

可以调用 Command#getMetrics 获取采样统计，然后通过 HystrixCommandMetrics 相关方法获取统计数据。

```
getExecutionTimePercentile(50);//P50
getExecutionTimePercentile(99);//P99
getExecutionTimePercentile(999);//P999
```

也可以订阅 HystrixDashboardStream.getInstance() 进行统计。Hystrix 提供了 hystrix-dashboard 进行图形化展示。

接下来我们通过 Turbine + Hystrix-Dashboard 实现集群化的统计可视化。



首先，Hystrix 应用会暴露统计接口，然后 Turbine 会聚合这些统计数据，Hystrix Dashboard 会拉取聚合后的统计信息并展示到仪表盘上。

5. Hystrix 客户端添加暴露统计信息 Servlet

```
@Bean
public ServletRegistrationBean servletRegistrationBean() {
    return new ServletRegistrationBean(
                new HystrixMetricsStreamServlet(), "/hystrix.stream");
}
```

在我们的 Hystrix 客户端添加如上 spring-boot 代码配置，然后就可以访问如 http://127.0.0.1:9080/hystrix.stream 获取到统计数据。

6. 部署 Turbine

下载 Turbine WAR 包（本文使用的是 Turbine 1.0.0），部署到 Tomcat 中，然后修改 WEB-INF/classes/config.properties 配置，启动 Tomcat。

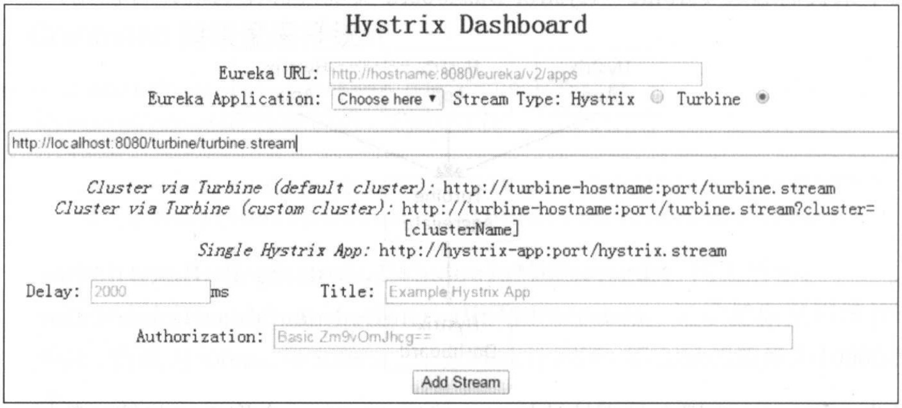
```
turbine.ConfigPropertyBasedDiscovery.default.instances=127.0.0.1
turbine.instanceUrlSuffix=:9080/hystrix.stream
```

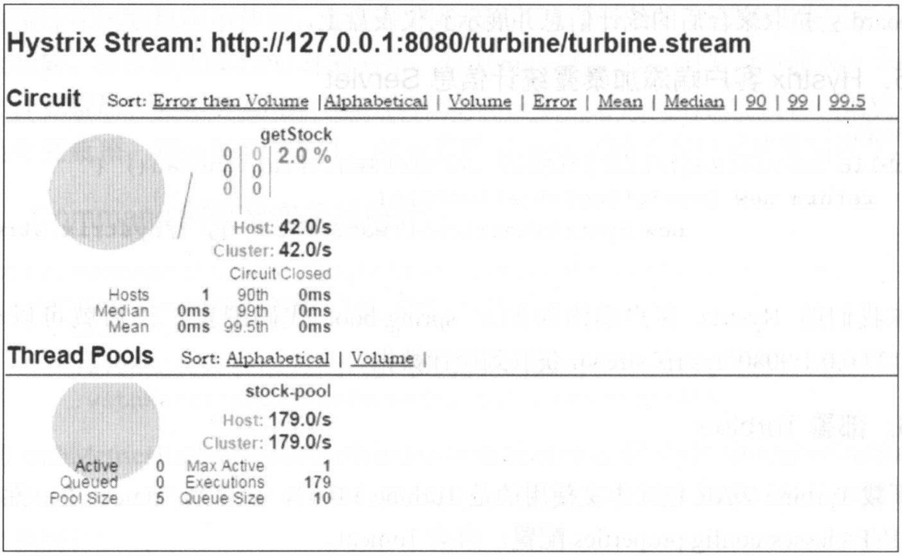
配置 Hystrix 应用的 IP 和获取统计信息的 URL path 部分，组合后拉取统计信息。访问如 http://127.0.0.1:8080/turbine/turbine.stream 获取聚合后的统计数据。

7．部署 Hystrix Dashboard

下载 hystrix-dashboard WAR 包（本文使用的是 hystrix-dashboard 1.5.6），部署到 Tomcat 中，然后启动 Tomcat。访问如 http://127.0.0.1:8080/hystrix-dashboard 启动仪表盘。

在如下界面添加要监控的 Turbine 地址，然后进入仪表盘就可以看到统计信息。



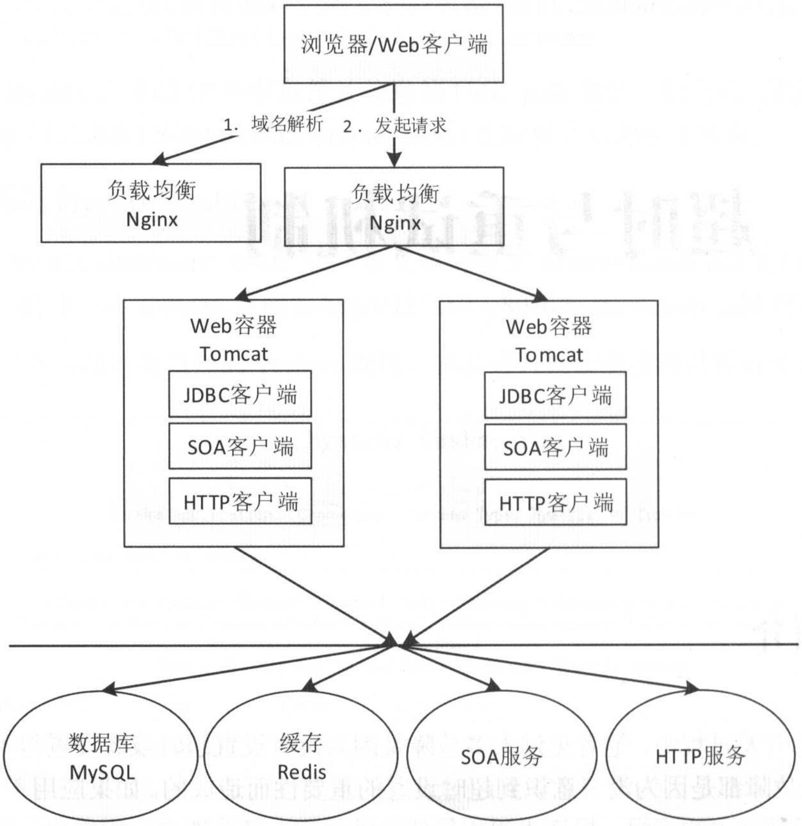


6 超时与重试机制

6.1 简介

在实际开发过程中，笔者见过太多故障是因为没有设置超时或者设置得不对而造成的。而这些故障都是因为没有意识到超时设置的重要性而造成的。如果应用不设置超时，则可能会导致请求响应慢，慢请求累积导致连锁反应，甚至造成应用雪崩。而有些中间件或者框架在超时后会进行重试（如设置超时重试两次），读服务天然适合重试，但写服务大多不能重试（如写订单，如果写服务是幂等的，则重试是允许的），重试次数太多会导致多倍请求流量，即模拟了 DDoS 攻击，后果可能是灾难，因此，务必设置合理的重试机制，并且应该和熔断、快速失败机制配合。在进行代码 Review 时，一定记得 Review 超时与重试机制。

本章主要从 Web 应用和服务化应用的角度出发介绍如何设置超时与重试（系统层面的超时设置在这里没有涉及），而 Web 应用需要在如下链条中设置超时与重试机制。



从上图来看，在整个链条中的每一个点都要考虑设置超时与重试机制。而其中最重要的超时设置是网络连接/读/写的超时时间设置。

下面将按照如下分类进行超时与重试机制的讲解。

- **代理层超时与重试**：如Haproxy、Nginx、Twemproxy，这些组件可实现代理功能，如Haproxy和Nginx可以实现请求的负载均衡。而Twemproxy可以实现Redis的分片代理。需要设置代理与后端真实服务器之间的网络连接/读/写超时时间。
- **Web容器超时**：如Tomcat、Jetty等，提供HTTP服务运行环境的，需要设置客户端与容器之间的网络连接/读/写超时时间，和在此容器中默认Socket网络连接/读/写超时时间。
- **中间件客户端超时与重试**：如JSF（京东SOA框架）、Dubbo、JMQ（京东消息中间件）、CXF、Httpclient等，需要设置客户的网络连接/读/写超时时间与失败重试机制。

- **数据库客户端超时**：如MySQL、Oracle，需要分别设置JDBC Connection、Statement的网络连接/读/写超时时间，事务超时时间，获取连接池连接等待时间。
- **NoSQL客户端超时**：如Mongo、Redis，需要设置其网络连接/读/写超时时间，获取连接池连接等待时间。
- **业务超时**：如订单取消任务、超时活动关闭，还有如通过Future#get(timeout, unit)限制某个接口的超时时间。
- **前端Ajax超时**：浏览器通过Ajax访问时的网络连接/读/写超时时间。

从如上分类可以看出，其中最重要的超时设置是网络相关的超时设置。

6.2 代理层超时与重试

对于代理层我们以 Nginx 和 Twemproxy 案例来讲解。首先，看一下 Nginx 的相关超时设置。

6.2.1 Nginx

Nginx 主要有 4 类超时设置：客户端超时设置、DNS 解析超时设置、代理超时设置，如果使用 ngx_lua，则还有 Lua 相关的超时设置。

1. 客户端超时设置

对于客户端超时主要设置有读取请求头超时时间、读取请求体超时时间、发送响应超时时间、长连接超时时间。通过客户端超时设置避免客户端恶意或者网络状况不佳造成连接长期被占用，影响服务器端的可处理能力。

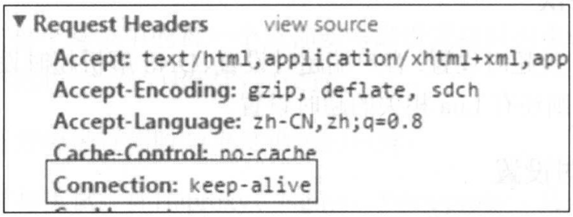
- **client_header_timeout time**：设置读取客户端请求头超时时间，默认为60s，如果在此超时时间内客户端没有发送完请求头，则响应408（Request Time-out）状态码给客户端。
- **client_body_timeout time**：设置读取客户端内容体超时时间，默认为60s，此超时时间指的是两次成功读操作间隔时间，而不是发送整个请求体的超时时间，

如果在此超时时间内客户端没有发送任何请求体，则响应408（Request Time-out）状态码给客户端。

- **send_timeout time**：设置发送响应到客户端的超时时间，默认为60s，此超时时间指的也是两次成功写操作间隔时间，而不是发送整个响应的超时时间。如果在此超时时间内客户端没有接收任何响应，则Nginx关闭此连接。
- **keepalive_timeout timeout [header_timeout]**：设置HTTP长连接超时时间。其中，第一个参数timeout是告诉Nginx长连接超时时间是多少，默认为75s。第二个参数header_timeout用于设置响应头“Keep-Alive: timeout=time”，即告知客户端长连接超时时间。两个参数可以不一样，“Keep-Alive: timeout=time”响应头可以在Mozilla和Konqueror系列浏览器中起作用，而MSIE长连接默认大约为60s，而不会使用“Keep-Alive: timeout=time”。如Httpclient框架会使用“Keep-Alive: timeout=time”响应头的超时（如果不设置默认，则认为是永久）。如果timeout设置为0，则表示禁用长连接。

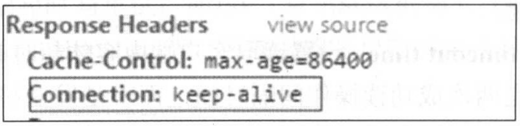
此参数要配合 keepalive_disable 和 keepalive_requests 一起使用。keepalive_disable 表示禁用哪些浏览器的长连接，默认值为 msie6，即禁用一些老版本的 MSIE 的长连接支持。keepalive_requests 参数的作用是一个客户端可以通过此长连接的请求次数，默认为 100。

首先，浏览器在请求时会通过如下请求头告知服务器是否支持长连接。

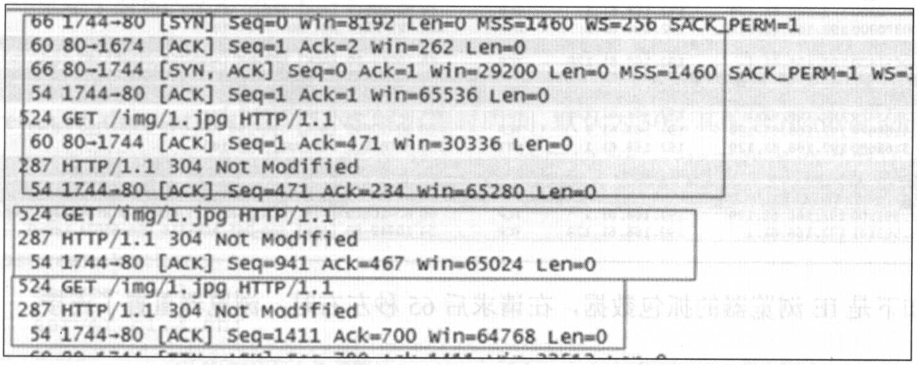


http/1.0 默认是关闭长连接的，需要添加 HTTP 请求头“Connection: keep-alive”才能启用。而 http/1.1 默认启用长连接，需要添加 HTTP 请求头“Connection: close”进行关闭。

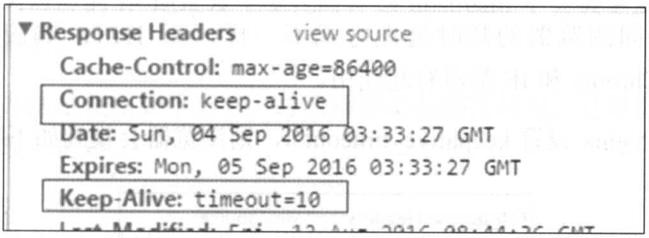
接着，如果 Nginx 设置 keepalive_timeout 5s，则浏览器会收到如下响应头。



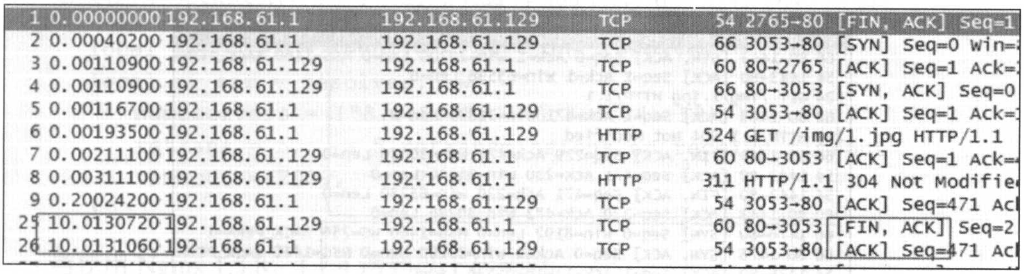
下图是 Wireshark 抓包，可以看到后两次请求没有三次握手。



如果 Nginx 设置 keepalive_timeout 10s 10s，则浏览器会收到如下响应头。

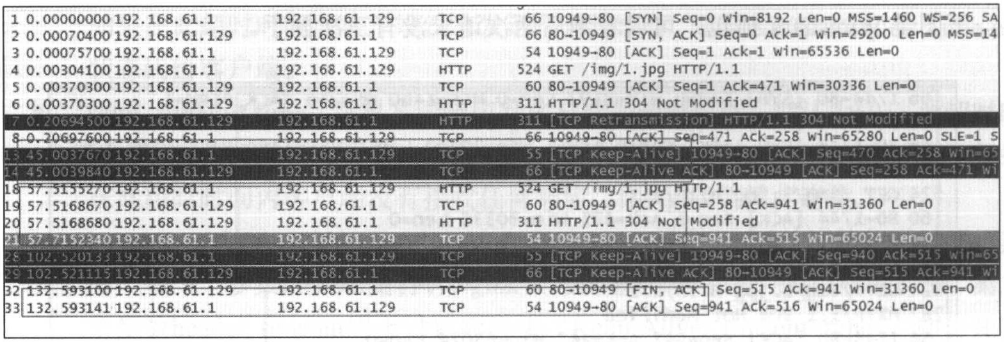


服务器端会在 10s 后发送 FIN 主动关闭连接。



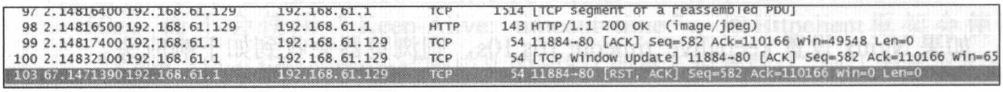
如果 Nginx 设置 keepalive_timeout 75s 30s。

如下是 Chrome 浏览器的 Wireshark 抓包。在 45s 时，Chrome 发送了 TCP Keep-Alive 来保持 TCP 连接。在 57s 时，浏览器又发出了一次请求。而在 132s 时，Nginx 发出了 FIN 来关闭连接（75s 后连接超时了）。



```
1 0.00000000 192.168.61.1    192.168.61.129  TCP   66 10949→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SA
2 0.00070400 192.168.61.129  192.168.61.1    TCP   66 80→10949 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=14
3 0.00075700 192.168.61.1    192.168.61.129  TCP   54 10949→80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4 0.00304100 192.168.61.1    192.168.61.129  HTTP  524 GET /img/1.jpg HTTP/1.1
5 0.00370300 192.168.61.129  192.168.61.1    TCP   60 80→10949 [ACK] Seq=1 Ack=471 Win=30336 Len=0
6 0.00370300 192.168.61.129  192.168.61.1    HTTP  311 HTTP/1.1 304 Not Modified
7 0.20694500 192.168.61.129  192.168.61.1    HTTP  311 [TCP Retransmission] HTTP/1.1 304 Not Modified
8 0.20697600 192.168.61.1    192.168.61.129  TCP   66 10949→80 [ACK] Seq=471 Ack=258 Win=65280 Len=0 SLE=1 S
13 45.0037670 192.168.61.1   192.168.61.129  TCP   55 [TCP Keep-Alive] 10949→80 [ACK] Seq=470 Ack=258 Win=65
14 45.0039840 192.168.61.129 192.168.61.1    TCP   66 [TCP Keep-Alive ACK] 80→10949 [ACK] Seq=258 Ack=471 Wi
18 57.5155270 192.168.61.1   192.168.61.129  HTTP  524 GET /img/1.jpg HTTP/1.1
19 57.5168670 192.168.61.129 192.168.61.1    TCP   60 80→10949 [ACK] Seq=258 Ack=941 Win=31360 Len=0
20 57.5168680 192.168.61.129 192.168.61.1    HTTP  311 HTTP/1.1 304 Not Modified
21 57.7152340 192.168.61.1   192.168.61.129  TCP   54 10949→80 [ACK] Seq=941 Ack=515 Win=65024 Len=0
28 102.520133 192.168.61.1   192.168.61.129  TCP   55 [TCP Keep-Alive] 10949→80 [ACK] Seq=940 Ack=515 Win=65
29 102.521115 192.168.61.129 192.168.61.1    TCP   66 [TCP Keep-Alive ACK] 80→10949 [ACK] Seq=515 Ack=941 Wi
32 132.593100 192.168.61.129 192.168.61.1    TCP   60 80→10949 [FIN, ACK] Seq=515 Ack=941 Win=31360 Len=0
33 132.593141 192.168.61.1   192.168.61.129  TCP   54 10949→80 [ACK] Seq=941 Ack=516 Win=65024 Len=0
```

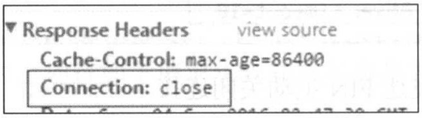
如下是 IE 浏览器的抓包数据，在请求后 65 秒左右时，浏览器重置了连接。

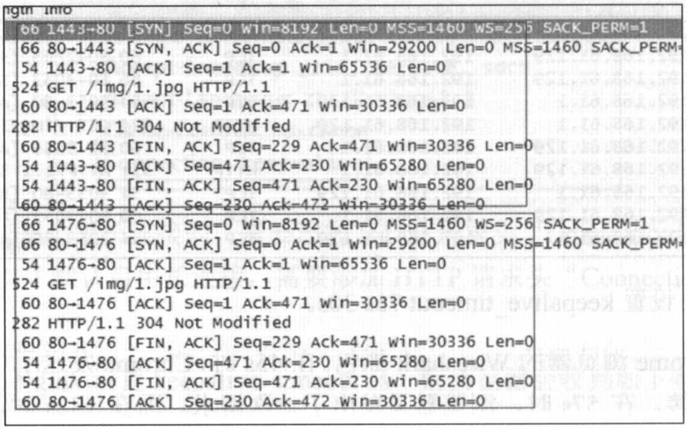


```
97 2.14816400 192.168.61.129  192.168.61.1    TCP   1514 [TCP segment of a reassembled PDU]
98 2.14816500 192.168.61.129  192.168.61.1    HTTP  143 HTTP/1.1 200 OK (image/jpeg)
99 2.14817400 192.168.61.1    192.168.61.129  TCP   54 11884→80 [ACK] Seq=582 Ack=110166 Win=49548 Len=0
100 2.14832100 192.168.61.1   192.168.61.129  TCP   54 [TCP Window Update] 11884→80 [ACK] Seq=582 Ack=110166 Win=65
103 67.1471390 192.168.61.1   192.168.61.129  TCP   54 11884→80 [RST, ACK] Seq=582 Ack=110166 Win=0 Len=0
```

可以看出不同浏览器的超时处理方式不一样，而 HTTP 响应头“Keep-Alive: timeout=30”对 Chrome 和 IE 都没有起作用。

接着，如果 Nginx 设置 keepalive_timeout 0，则浏览器会收到如下响应头。





```
ngth Info
66 1443→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
66 80→1443 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM
54 1443→80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
524 GET /img/1.jpg HTTP/1.1
60 80→1443 [ACK] Seq=1 Ack=471 Win=30336 Len=0
282 HTTP/1.1 304 Not Modified
60 80→1443 [FIN, ACK] Seq=229 Ack=471 Win=30336 Len=0
54 1443→80 [ACK] Seq=471 Ack=230 Win=65280 Len=0
54 1443→80 [FIN, ACK] Seq=471 Ack=230 Win=65280 Len=0
60 80→1443 [ACK] Seq=230 Ack=472 Win=30336 Len=0
66 1476→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
66 80→1476 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM
54 1476→80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
524 GET /img/1.jpg HTTP/1.1
60 80→1476 [ACK] Seq=1 Ack=471 Win=30336 Len=0
282 HTTP/1.1 304 Not Modified
60 80→1476 [FIN, ACK] Seq=229 Ack=471 Win=30336 Len=0
54 1476→80 [ACK] Seq=471 Ack=230 Win=65280 Len=0
54 1476→80 [FIN, ACK] Seq=471 Ack=230 Win=65280 Len=0
60 80→1476 [ACK] Seq=230 Ack=472 Win=30336 Len=0
```

对于客户端超时设置，要根据实际场景来决定。如果是短连接服务，则可以考虑将时间设置得短一些，如果是文件上传，则需要考虑设置得长一些。另外，笔者见过很多人对长连接没有正确配置，建议配置完成后通过抓包查看长连接是否起作用。

keepalive_timeout 和 keepalive_requests 是控制长连接的两个维度，只要其中一个到达设置的阈值，连接就会被关闭。

2. DNS 解析超时设置

resolver_timeout 30s: 设置 DNS 解析超时时间，默认为 30s。其配合 resolver address ... [valid=time]进行 DNS 域名解析。当在 Nginx 中使用域名时，就需要考虑设置这两个参数。在 Nginx 社区版中采用如下配置。

```
upstream backend {
   server c0.3.cn;
   server c1.3.cn;
}
```

如上两个域名会在 Nginx 解析配置文件的阶段被解析成 IP 地址并记录到 upstream 上，当这两个域名对应的 IP 地址发生变化时，该 upstream 不会被更新。而 Nginx 商业版是支持动态更新的。

一种简单的办法是使用如下方式，每次都会动态解析域名，这种情况在多域名情况下比较麻烦，实现不优雅。

```
location /test {
   proxy_pass http://c0.3.cn;
}
```

如果使用 OpenResty，则可以通过 Lua 库 lua-resty-dns 进行 DNS 解析。

```
local resolver = require "resty.dns.resolver"
local r, err = resolver:new{
   nameservers = {"8.8.8.8", {"8.8.4.4", 53} },
   retrans = 5,  -- 5 retransmissions on receive timeout
   timeout = 2000,  -- 2 sec
}
```

当使用 Nginx 1.5.8、1.7.4 遇到以下代码时，

```
could not be resolved(110:Operation timed out);
```

或者

```
wrong ident 37278 response for ***.jd.local, expected 33517
unexpected response for ***.jd.local
```

可能是遇到了如下 BUG（http://nginx.org/en/CHANGES-1.6、http://nginx.org/en/

CHANGES-1.8）。

```
Bugfix: requests might hang if resolver was used and a timeout
     occurred during a DNS request.
```

请考虑升级到 Nginx 1.6.2、1.7.5 或者在 Nginx 本机部署 dnsmasq 提升 DNS 解析性能。

3. 代理超时设置

Nginx 配置如下所示。

```
upstream backend_server {
    server 192.168.61.1:9080 max_fails=2 fail_timeout=10s weight=1;
    server 192.168.61.1:9090 max_fails=2 fail_timeout=10s weight=1;
}
server {
    ......
    location /test {
        proxy_connect_timeout 5s;
        proxy_read_timeout 5s;
        proxy_send_timeout 5s;

        proxy_next_upstream error timeout;
        proxy_next_upstream_timeout 0;
        proxy_next_upstream_tries 0;

        proxy_pass http://backend_server;
        add_header upstream_addr $upstream_addr;
    }
}
```

backend_server 定义了两个上游服务器 192.168.61.1:9080（返回 hello）和 192.168.61.1:9090（返回 hello2）。

如上指令主要有三组配置：网络连接/读/写超时设置、失败重试机制设置、upstream 存活超时设置。

① 网络连接/读/写超时设置

- **proxy_connect_timeout time：**与后端/上游服务器建立连接的超时时间，默认为60s，此时间不超过75s。
- **proxy_read_timeout time：**设置从后端/上游服务器读取响应的超时时间，默认为60s，此超时时间指的是两次成功读操作间隔时间，而不是读取整个响应体

的超时时间，如果在此超时时间内上游服务器没有发送任何响应，则Nginx关闭此连接。

- **proxy_send_timeout time**：设置往后端/上游服务器发送请求的超时时间，默认为60s，此超时时间指的是两次成功写操作间隔时间，而不是发送整个请求的超时时间，如果在此超时时间内上游服务器没有接收任何响应，则Nginx关闭此连接。

对于内网高并发服务，请根据需要调整这几个参数，比如内网服务 TP999 为 1s，可以将连接超时设置为 100~500ms，而读超时可以为 1.5~3s 左右。

② 失败重试机制设置

- **proxy_next_upstream error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 |http_403 | http_404 | non_idempotent | off ...**：配置什么情况下需要请求下一台上游服务器进行重试。默认为“error timeout”。error表示与上游服务器建立连接、写请求或者读响应头出错。timeout表示与上游服务器建立连接、写请求或者读响应头超时。invalid_header表示上游服务器返回空的或错误的响应头。http_XXX表示上游服务器返回特定的状态码。non_idempotent表示RFC-2616定义的非幂等HTTP方法（POST、LOCK、PATCH），也可以在失败后重试下一台上游服务器（即默认幂等方法GET、HEAD、PUT、DELETE、OPTIONS、TRACE才可以重试）。off表示禁用重试。

重试不能无限制进行，因此，需要如下两个指令控制重试次数和重试超时时间。

- **proxy_next_upstream_tries number**：设置重试次数，默认0表示不限制，注意此重试次数指的是所有请求次数（包括第一次和之后的重试次数之和）。
- **proxy_next_upstream_timeout time**：设置重试最大超时时间，默认0表示不限制。

即在 proxy_next_upstream_timeout 时间内允许 proxy_next_upstream_tries 次重试。如果超过了其中一个设置，则 Nginx 也会结束重试并返回客户端响应（可能是错误码）。

如下配置表示当 error/timeout 时重试 upstream 中的下一台上游服务器，如果重试的总时间超过 6s 或者重试了 1 次，则表示重试失败（因为之前已经请求一次了，所以还能重试 1 次），Nginx 结束重试并返回客户端响应。

```
proxy_next_upstream error timeout;
proxy_next_upstream_timeout 6s;
proxy_next_upstream_tries 2;
```

③ upstream 存活超时设置

- **max_fails和fail_timeout：**配置什么时候Nginx将上游服务器认定为不可用/不存活。当上游服务器在fail_timeout时间内失败了max_fails次，则认为该上游服务器不可用/不存活。并在接下来的fail_timeout时间内从upstream摘掉该节点（即请求不会转发到该上游服务器）。

什么情况下被认定为失败呢？其由 proxy_next_upstream 定义，不过，不管 proxy_next_upstream 如何配置，error, timeout 和 invalid_header 都将被认为是失败。

如 server 192.168.61.1:9090 max_fails=2 fail_timeout=10s;表示在 10s 内如果失败了 2 次，则在接下来的 10s 内认定该节点不可用/不存活。这种存活检测机制只有当访问该上游服务器时，采取惰性检查，才可以使用 ngx_http_upstream_check_module 配置主动检查。

max_fails 设置为 0 表示不检查服务器是否可用（即认为一直可用），如果 upstream 中仅剩一台上游服务器，则该服务器是不会被摘除的，将从不被认为不可用。

④ ngx_lua 超时设置

当我们使用 ngx_lua 时，也应考虑设置如下网络连接/读/写超时。

```
lua_socket_connect_timeout  100ms;
lua_socket_send_timeout    200ms;
lua_socket_read_timeout    500ms;
```

在使用 Lua 时，我们会按照如下策略进行重试。

```
   if (status == 502 or status == 503 or status == 504) and request_time <
200 then
      resp = capture(proxy_uri)
      status = resp.status
      body = resp.body
      request_time = request_time + tonumber(var.request_time) * 1000
   end
```

即如果状态码是 500/502/503/504，并且该次请求耗时在 200ms 以内，则我们进行一次重试。

6.2.2 Twemproxy

Twemproxy 是 Twitter 开源的 Redis 和 Memcache 代理中间件，其目的是减少与后端

缓存服务器的连接数。

timeout：表示与后端服务器建立连接、接收响应的超时时间，默认永不超时。

server_retry_timeout 和 server_failure_limit：当开启 auto_eject_hosts，即当后端服务器不可用时自动摘除这些节点并在一定时间后进行重试。server_failure_limit 设置连续失败多少次后将节点临时摘除，server_retry_timeout 设置摘除节点后等待多久进行重试，从而保证不永久性地将节点摘除。

6.3 Web 容器超时

笔者的生产环境用的 Java Web 容器是 Tomcat，本部分将以 Tomcat 8.5 作为例子进行讲解。

- **connectionTimeout**：配置与客户端建立连接的超时时间，从接收到连接后，在配置的时间内没有接收到客户端请求行，将被认定为连接超时，默认为60000（60s）。
- **socket.soTimeout**：从客户端读取请求数据的超时时间，默认同 connectionTimeout，NIO和NIO2支持该配置。
- **asyncTimeout**：Servlet 3异步请求的超时时间，默认为30000（30s）。
- **disableUploadTimeout和connectionUploadTimeout**：当配置disableUploadTimeout为 false 时（默认为 true，和 connectionTimeout 一样），文件上传将使用connectionUploadTimeout作为超时时间。
- **keepAliveTimeout和maxKeepAliveRequests**：和Nginx配置类似。keepAliveTimeout默认为connectionTimeout，配置为-1表示永不超时。maxKeepAliveRequests默认为100。

6.4 中间件客户端超时与重试

JSF 是京东自研的 SOA 框架，主要有三个组件：注册中心、服务提供端、服务消费端。

首先是在服务提供端/消费端与注册中心之间进行服务注册/发现时可以配置 timeout（调用注册中心超时时间，默认为 5s）和 connectTimeout（连接注册中心的超时时间，默认为 20s）。

服务提供端可以配置 timeout（服务器端调用超时时间，默认为 5s）。

服务消费端可以配置 timeout（调用端调用超时时间，默认为 5s）、connectTimeout（建立连接超时时间，默认为 5s）、disconnectTimeout（断开连接/等待结果超时时间，默认为 10s）、reconnect（调用端重连死亡服务器端的间隔，配置小于 0 表示不重连，默认为 10s）、heartbeat（调用端往服务器端发心跳包的间隔，配置小于 0 代表不发送，默认为 30s）和 retries（失败后重试次数，默认 0 不重试）。

Dubbo 也有类似的配置，在此就不赘述了。

JMQ 是京东消息中间件，主要有四个组件：注册中心、Broker（JMQ 的服务器端实例，生产和消费消息都跟它交互）、生产者、消费者。

首先是在生产者/消费者与 Broker 进行发送/接收消息时，可以配置 connectionTimeout（连接超时）、sendTimeout（发送超时）和 soTimeout（读超时）。

生产者可以配置 retryTimes（发送失败后的重试次数，默认为 2 次）。

消费者可以配置 pullTimeout（长轮询超时时间，即拉取消息超时时间）、maxRetrys（最大重试次数，对于消费者要允许无限制重试，即一直拉取消息）、retryDelay（重试延迟，通过 exponential 配置延迟增加倍数一直增加到 maxRetryDelay）、maxRetryDelay（最大重试延迟）。消费者还需要配置应答超时时间（服务器端需要等待客户端返回应答才能移除消息，如果没有应答返回，则会等待应答超时，在这段时间内锁定的消息不能被消费，必须等待超时后才能被消费）。

对于消息中间件，我们在实际应用中关注超时配置会少一些，因为生产者默认配置了重试次数，可能会存在重复消息，消费者需要进行去重处理。

CXF 可以通过如下方式配置 CXF 客户端连接超时、等待响应超时和长连接。

```
HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();
httpClientPolicy.setConnectionTimeout(30000);//默认为 30s
httpClientPolicy.setReceiveTimeout(60000); //默认为 60s
httpClientPolicy.setConnection(ConnectionType.KEEP_ALIVE);//默认为 Keep-
Alive
((HTTPConduit)client.getConduit()).setClient(httpClientPolicy);
```

Httpclient 4.2.x 可以通过如下代码配置网络连接、等待数据超时时间。

```
HttpParams params = new BasicHttpParams();
//设置连接超时时间
Integer CONNECTION_TIMEOUT = 2 * 1000;    //设置请求超时 2s
Integer SO_TIMEOUT = 2 * 1000;    //设置等待数据超时时间 2s
Long CONN_MANAGER_TIMEOUT = 1L * 1000;    //定义了当从
//ClientConnectionManager 中检索 ManagedClientConnection
//实例时使用的毫秒级的超时时间
params.setIntParameter(CoreConnectionPNames.CONNECTION_TIMEOUT,
CONNECTION_TIMEOUT);
params.setIntParameter(CoreConnectionPNames.SO_TIMEOUT, SO_TIMEOUT);
//在提交请求之前，测试连接是否可用
params.setBooleanParameter(CoreConnectionPNames.STALE_CONNECTION_CHECK,
true);
//这个参数期望得到一个 java.lang.Long 类型的值。如果这个参数没有被设置，
//则连接请求就不会超时（无限大的超时时间）
params.setLongParameter(ClientPNames.CONN_MANAGER_TIMEOUT,
CONN_MANAGER_TIMEOUT);
PoolingClientConnectionManager conMgr = new PoolingClientConnectionManager();
conMgr.setMaxTotal(200);//设置最大连接数
//是路由的默认最大连接（该值默认为 2），限制数量实际使用 DefaultMaxPerRoute
//而非 MaxTotal
//设置过小，无法支持大并发(ConnectionPoolTimeoutException: Timeout waiting
//for connection from pool)，路由是对 maxTotal 的细分
conMgr.setDefaultMaxPerRoute(conMgr.getMaxTotal());
//（目前只有一个路由，因此让它等于最大值）

//设置访问协议
conMgr.getSchemeRegistry().register(new Scheme("http", 80, PlainSocketFactory.
getSocketFactory()));
conMgr.getSchemeRegistry().register(new Scheme("https", 443, SSLSocketFactory.
getSocketFactory()));
```

```
    httpClient = new DefaultHttpClient(conMgr, params);
    httpClient.setHttpRequestRetryHandler(new
DefaultHttpRequestRetryHandler(0, false));
```

因为我们使用 http connection 连接池，所以需要配置 CONN_MANAGER_TIMEOUT，表示从连接池获取 http connection 的超时时间。

此处还通过 httpClient.setHttpRequestRetryHandler(new DefaultHttpRequestRetryHandler(0, false))配置了请求重试策略（默认重试 3 次）。当执行请求遇到异常时，会调用 retryRequest 来判断是否进行重试，而以下情况不会进行重试：达到重试次数、服务器不可达、连接被拒绝、连接终止、请求已发送。而幂等 HTTP 方法的请求、requestSentRetryEnabled=true 且请求还未成功发送时可以重试。

如果响应 503 错误状态码，如上重试机制是不可用的，则可以考虑使用 AutoRetryHttpClient 客户端，其可以配置 ServiceUnavailableRetryStrategy，默认实现为 DefaultServiceUnavailableRetryStrategy，可以配置重试次数 maxRetries 和重试间隔 retryInterval。每次重试之前都会等待 retryInterval 毫秒时间。

假设服务由多个机房提供，其中在一个机房服务出现问题时，应该自动切换到另一个机房，可以考虑使用如下方法。

```
    public static String get(List<String> apis, Object[] args, String encoding,
Header[] headers, Integer timeout) throws Exception {
        String response = null;
        for(String api : apis) {
          String uri = UriComponentsBuilder.fromHttpUrl(api)
                                  .buildAndExpand(args). toUriString();
            response = HttpClientUtils
                        .getDataFromUri(uri, encoding, headers, timeout);
            //如果失败了，重试一次
            if(Objects.equal(response, HTTP_ERROR)) {
                continue;
            }
            //如果域名解析失败，重试
            if(Objects.equal(response, HTTP_UNKNOWN_HOST_ERROR)) {
                response = HTTP_ERROR; //调用方根据这个判断是否有问题
                continue;
            }
            if(Objects.equal(response, HTTP_SOCKET_TIMEOUT_ERROR)) {
                response = HTTP_ERROR; //调用方根据这个判断是否有问题
                continue;
            }
```

```
        return response;
    }
    return response;
}
```

参数传入不同机房的 API 即可，当其中一个不可用时自动重试另一个机房的 API。

6.5 数据库客户端超时

在使用数据库客户端时，我们会使用数据库连接池，数据库连接池可以进行如下超时设置。

```
<bean id="dataSource"
      class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close">
    <!-- Statement 默认超时时间 -->
    <property name="defaultQueryTimeout" value="3"/>

    <!-- 另外可以通过如下配置来配置 socket 连接/读超时: -->
    <property name="connectionProperties"
              value="connectTimeout=2000; socketTimeout=2000 "/>
    <!-- 这个是等待获取连接池连接时间，也不要太大，比如设置在 500ms -->
    <property name="maxWaitMillis" value="500" />
</bean>
```

- 网络连接/读超时：使用connectionProperties配置MySQL超时时间，如果是Oracle则可以通过如下配置。

```
<property name="connectionProperties"
value="oracle.net.CONNECT_TIMEOUT=2000;oracle.jdbc.ReadTimeout=2000"/>
```

- 默认Statement超时时间，通过defaultQueryTimeout配置，单位是s。
- 从连接池获取连接的等待时间，通过maxWaitMillis配置。
- Statement超时，如果使用iBATIS，则可以通过如下方式配置Statement超时。

```
<settings cacheModelsEnabled="false" enhancementEnabled="true"
          lazyLoadingEnabled="false" errorTracingEnabled="true"
          maxRequests="32" defaultStatementTimeout="2"/>
```

defaultStatementTimeout 的单位是 s，根据业务配置。如果配置了数据库连接池，则此处不用配置。

如果只想设置某个 Statement 的超时时间，则可以考虑<insert ……timeout="2">。

如上配置其实最终会调用 Statement.setQueryTimeout 方法设置 Statement 超时时间。

- 事务超时是总的Statement超时设置，比如我们使用Spring管理事务，可以使用如下方式配置全局默认的事务级别的超时时间。

```
<bean id="txManager" class="org.springframework.jdbc.datasource
                                   .DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
  <property name="defaultTimeout" value="3"/>
</bean>
```

这里我们分析为什么说事务超时是 Statement 超时的总和，此处我们分析 Spring 的 DataSourceTransactionManager，首先开启事务时会调用其 doBegin 方法。

```
//先获取@Transactional 定义的 timeout，如果没有，则使用 defaultTimeout
int timeout = determineTimeout(definition);
if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
   txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
}
```

其中 determineTimeout 用来获取我们设置的事务超时时间，然后设置到 ConnectionHolder 对象上（其是 ResourceHolder 子类），接着下面看 ResourceHolderSupport 的 setTimeoutInSeconds 实现。

```
public void setTimeoutInSeconds(int seconds) {
   setTimeoutInMillis(seconds * 1000);
}

public void setTimeoutInMillis(long millis) {
   this.deadline = new Date(System.currentTimeMillis() + millis);
}
```

大家可以看到，此处会设置一个 deadline 时间，用来判断事务超时时间，那什么时候调用呢？首先检查该类中的代码：

```
public int getTimeToLiveInSeconds() {
   double diff = ((double) getTimeToLiveInMillis()) / 1000;
   int secs = (int) Math.ceil(diff);
   checkTransactionTimeout(secs <= 0);
   return secs;
```

```
    }

    public long getTimeToLiveInMillis() throws TransactionTimedOutException{
    if (this.deadline == null) {
        throw new IllegalStateException("No timeout specified for this 
resource holder");
    }
    long timeToLive = this.deadline.getTime() - System.currentTimeMillis();
        checkTransactionTimeout(timeToLive <= 0);
        return timeToLive;
    }

    private void checkTransactionTimeout(boolean deadlineReached)
                                        throws TransactionTimedOutException {
        if (deadlineReached) {
            setRollbackOnly();
            throw new TransactionTimedOutException("Transaction timed out: 
deadline was " + this.deadline);
        }
    }
```

我们发现调用 getTimeToLiveInSeconds 和 getTimeToLiveInMillis 会检查是否超时，如果超时了，则标记事务须回滚，并抛出 TransactionTimedOutException 异常进行回滚。

DataSourceUtils.applyTransactionTimeout 会调用 DataSourceUtils. applyTimeout, DataSourceUtils.applyTimeout 的代码如下。

```
    public static void applyTimeout(Statement stmt, DataSource dataSource,
                                    int timeout) throws SQLException {
        ConnectionHolder holder =
            (ConnectionHolder) TransactionSynchronizationManager
                                            .getResource(dataSource);
        if (holder != null && holder.hasTimeout()) {
            // 计算剩余的事务超时时间并覆盖 Statement 超时
            stmt.setQueryTimeout(holder.getTimeToLiveInSeconds());
        } else if (timeout > 0) {
            //如果没有配置事务超时，则使用 Statement 超时
            stmt.setQueryTimeout(timeout);
        }
    }
```

在执行 stmt.setQueryTimeout(holder.getTimeToLiveInSeconds())时会调用 getTimeToLiveIn Seconds()，这会检查事务是否超时。在 JdbcTemplate 中，执行 SQL 之前，会调用其 applyStatementSettings 方法，其将调用 DataSourceUtils.applyTimeout(stmt,

getDataSource(), getQueryTimeout())设置超时时间。

此处有一个问题，如果设置了事务超时，Statement 超时的就不起作用了，整体会使用事务超时覆盖 Statement 超时。

6.6 NoSQL 客户端超时

对于 MongoDB，我们使用的是 spring-data-mongodb 客户端，可以通过如下配置设置相关的超时时间。

```
<mongo:mongo id="tryMongo" replica-set="${try.mongo.hostAndPorts}">
   <mongo:options
          connections-per-host="${mongo.connectionsPerHost}"
          threads-allowed-to-block-for-connection-multiplier=
               "${mongo.threadsAllowedToBlockForConnectionMultiplier}"
          max-wait-time="${mongo.maxWaitTime}"
          connect-timeout="${mongo.connectTimeout}"
          socket-timeout="${mongo.socketTimeout}"
          socket-keep-alive="${mongo.socketKeepAlive}"
          auto-connect-retry="${mongo.autoConnectRetry}" />
</mongo:mongo>
```

我们曾经就遇到过因为不设置 MongoDB 客户端超时而导致服务响应慢的情况。

对于 Redis，我们使用的是 Jedis 客户端，可以通过如下配置分配等待获取连接池连接的超时时间和网络连接/读超时时间。

```
PoolJedisConnectionFactory connectionFactory = new PoolJedisConnectionFactory();
connectionFactory.setMaxWaitMillis(maxWaitMillis);
connectionFactory.setTimeout(timeoutInMillis);
```

Jedis 在建立 Socket 时通过如下代码设置超时。

```
this.socket.connect(
        new InetSocketAddress(this.host, this.port), this. timeout);
this.socket.setSoTimeout(this.timeout);
```

可以在 JVM 启动时通过添加 -Dsun.net.client.defaultConnectTimeout=60000

-Dsun.net.client.defaultReadTimeout=60000 来配置默认的全局 Socket 连接/读超时。即如 Httpclient、JDBC 等，如果没有配置 Socket 超时，则会默认使用该超时。

6.7 业务超时

业务超时分为如下两类。

- **任务型：**比如，订单超时未支付取消超时活动自动关闭等，这属于任务型超时，可以通过Worker定期扫描数据库修改状态。有时需要调用的远程服务超时了（比如，用户注册成功后，需要给用户发放优惠券），可以考虑使用队列或者暂时记录到本地稍后重试。
- **服务调用型：**比如，某个服务的全局超时时间为500ms，但我们有多处服务调用，每处服务调用的超时时间可能不一样，此时，可以简单地使用Future来解决问题，通过如Future.get(3000, TimeUnit.MILLISECONDS)来设置超时。

6.8 前端 Ajax 超时

我们使用 jQuery 来进行 Ajax 请求，可以在请求时带上 timeout 参数设置超时时间。

```
$.ajax({
   url:"http://ins.jd.com:9090/test",
   dataType:"jsonp",
   jsonp:"test",
   jsonpCallback:"test",
   timeout:2000,
   success: function(result,status,xhr) {
      //success
   },
   error: function(result,status,xhr) {
      if(status == 'timeout') {
```

```
                //timeout
            }
        }
    });
```

当进行跨域 JSONP 请求并使用 jQuery 1.4.x 版本时，IE9、Chrome 52、Firefox 49 测试 JSONP，请求在超时后不能被取消，即使客户端超时了，该脚本也将一直运行；而使用 jQuery 1.5.2 时，超时是起作用的，但是发出去的请求是不会被取消的（请求还处于执行状态）。

还有一种办法可进行超时重试，即通过 setTimeout 进行超时重试。比如，京东首页的某个异步接口，其中一个域名（A 机房）超时了，想超时后通过另一个域名（B 机房）重新获取数据，代码如下所示。

```
var id = setTimeout(retryCallback, 5000);
$.ajax({
    dataType: 'jsonp',
    success: function() {
        clearTimeout(id);
        ...
    }
});
```

除了客户端设置超时外，服务器端也一定要配置合理的超时时间。

6.9 总结

本章主要介绍了如何在 Web 应用访问的整个链路上进行超时时间设置。通过配置合理的超时时间，防止出现某服务的依赖服务超时时间太长且响应慢，以致自己响应慢甚至崩溃。

客户端和服务器端都应该设置超时时间，而且客户端根据场景可以设置比服务器端更长的超时时间。如果存在多级依赖关系，如 A 调用 B，B 调用 C，则超时设置应该是 A>B>C，否则可能会一直重试，引起 DDoS 攻击效果。不过最终如何选择还是要看场景，有时候客户端设置的超时时间就是要比服务器端的短，可以通过在服务器端实施限流/降级等手段防止 DDoS 攻击。

超时之后应该有相应的策略来处理，常见的策略有重试（等一会儿再试、尝试其他分组服务、尝试其他机房服务，重试算法可考虑使用如指数退避算法）、摘掉不存活节点（负载均衡/分布式缓存场景下）、托底（返回历史数据/静态数据/缓存数据）、等待页或者错误页。

对于非幂等写服务应避免重试，或者可以考虑提前生成唯一流水号来保证写服务操作通过判断流水号来实现幂等操作。

在进行数据库/缓存服务器操作时，记得经常检查慢查询，慢查询通常是引起服务出问题的罪魁祸首。也要考虑在超时严重时，直接将该服务降级，待该服务修复后再取消降级。

对于有负载均衡的中间件，请考虑配置心跳/存活检查，而不是惰性检查。

超时重试必然导致请求响应时间增加，最坏情况下的响应时间=重试次数×单次超时时间，这很可能严重影响用户体验，导致用户不断刷新页面来重复请求，最后导致服务接收的请求太多而挂掉，因此除了控制单次超时时间，也要控制好用户能忍受的最长超时时间。

超时时间太短会导致服务调用成功率降低，超时时间太长又会导致本应成功的调用却失败了，这也要根据实际场景来选择最适合当前业务的超时时间，甚至是程序动态自动计算超时时间。比如商品详情页的库存状态服务，可以设置较短的超时时间，当超时时降级返回有货，而结算页服务就需要设置稍微长一些的超时时间保证确实有货。

在实际开发中，不要轻视超时时间，很多重大事故都是因为超时时间不合理导致的，设置超时时间一定是只有好处没有坏处的，请立即 Review 你的代码吧。

6.10 参考资料

[1] https://github.com/twitter/twemproxy/blob/master/notes/recommendation.md#liveness

[2] http://nginx.org/en/docs/http/ngx_http_core_module.html

[3] http://nginx.org/en/docs/http/ngx_http_upstream_module.html

[4] http://nginx.org/en/docs/http/ngx_http_proxy_module.html

[5] https://github.com/openresty/lua-resty-dns

[6] http://tomcat.apache.org/tomcat-8.5-doc/config/http.html

[7] http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html

[8] http://jinnianshilongnian.iteye.com/blog/2302325

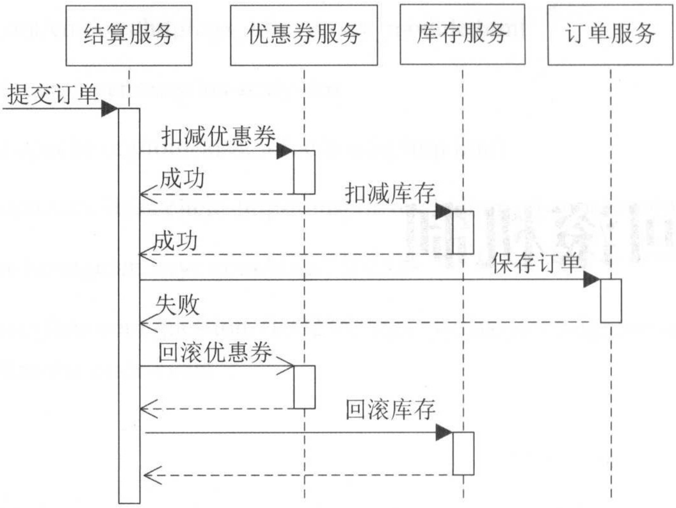
[9] http://stackoverflow.com/questions/1002367/jquery-ajax-jsonp-ignores-a-timeout-and-doesnt-fire-the-error-event

7 回滚机制

回滚是指当程序或数据出错时，将程序或数据恢复到最近的一个正确版本的行为。最常见的如事务回滚、代码库回滚、部署版本回滚、数据版本回滚、静态资源版本回滚等。通过回滚机制可保证系统在某些场景下的高可用。

7.1 事务回滚

在执行数据库 SQL 时，如果我们检测到事务提交冲突，那么事务中所有已执行的 SQL 要进行回滚，目的是防止数据库出现数据不一致。对于单库事务回滚直接使用相关 SQL 即可。如果涉及分布式数据库，则要考虑使用分布式事务，最常见的如两阶段提交、三阶段提交协议，这种方式实现事务回滚难度较低，但是对性能影响比较大，因为我们在大多数场景中需要的是最终一致性，而不是强一致性。因此，可以考虑如事务表、消息队列、补偿机制（执行/回滚）、TCC 模式（预占/确认/取消）、Sagas 模式（拆分事务+补偿机制）等实现最终一致性。比如，电商中的单场景，会进行扣减优惠券、预占库存等操作，这涉及非常多的子系统，因此，很难使用分布式事务保证强一致性，我们只要能保证最终一致性即可，下面来看看结算下单序列图。



一种情况是当订单出错后，要把之前扣减的优惠券和库存回滚。但是，当保存订单出错时，JVM 实例挂掉了，那么之前扣减的优惠券和库存就没有回滚，这种情况可以考虑在本地记录事务日志，当 JVM 实例重启后，分析事务日志重新回滚，当然也可以记录事务日志表，或者通过补偿机制，定期扫描优惠券和库存使用表，回滚没有关联订单的或者已取消订单的记录。还有一种情况是下单后一直没有支付，比如 6 小时，没有支付的订单要取消，此时就要定期扫描订单表，然后取消订单并回滚优惠券和库存。不管用什么方式，只要保证最终一致性即可。

7.2 代码库回滚

在开发项目时，一定要将代码维护到代码仓库，从而进行版本管理。常见的有 SVN、Git 等，SVN 是一款集中版本控制系统，而 Git 是一款分布式版本控制系统。有了版本控制系统后就可以记录代码的历史版本，在出问题后可以方便回滚。当某个代码文件部署出现问题时，可以通过历史版本查看是谁修改的、修改了什么，从而快速定位出 BUG。另外，在实际开发过程中，可能存在多个版本并行开发，此时版本控制系统的分支功能就发挥大作用了，大家在各自分支上开发测试，相互不影响，开发完成后合并分支到主干即可。

7.3 部署版本回滚

代码测试完成后，接下来就要进行系统的部署，在部署系统时，要考虑当代码逻辑出现错误后如何快速恢复，总结为部署版本化、小版本增量发布、大版本灰度发布、架构升级并发发布。

1. 部署版本化

每次部署时，应该将上一版本的包记录到部署系统中，在发布时应该采用全量发布，避免增量发布（只发布修改过的类或文件）。如有需要，全量版本可直接回滚，不会受到约束或限制。

2. 小版本增量发布

比如修复 BUG，添加一些简单的业务逻辑，这些我们叫作小版本。增量发布的意思是比如我们有 100 台服务器，先发布 1 台验证，如果没问题，则接着发布 10 台，最后全量发布。

3. 大版本灰度发布

在页面改版、添加新的功能时需要进行灰度发布，一般情况下是两个版本并行跑一段时间，一些用户访问老版本，一些用户访问新版本，功能验证成功后或者新版本效果不错时，再全量发布。比如，我们可以通过类似如下带有版本号的 URL 来区分新版本和老版本。

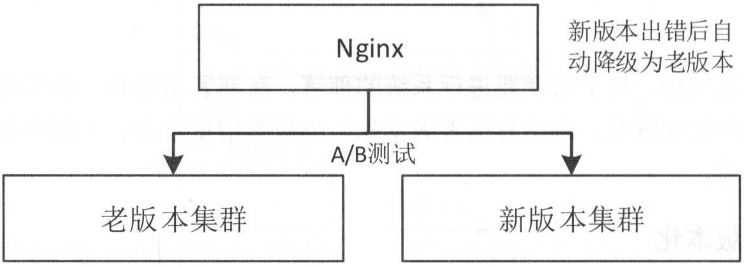
```
https://cd.jd.com/yanbao/v3?skuId=854073&cat=652,654,832&brandId=8983&
area=1_2810_51081_0&callback=yanbao_jsonp_callback
```

不同版本其实就是不同的服务，在一套集群部署即可，出问题时要能非常快速地切换回老版本。

4. 架构升级并发发布

架构升级后，我们不太清楚新版本是否功能正常，因此，新老版本部署集群会同时存在一段时间。然后，等所有流量迁移到新版本集群后，老版本集群就可以下线了。

一般前端应用我们会采用 Nginx 作为接入层，通过 A/B 方式慢慢地将流量引入到新版本集群，比如 1%→10%→50%→100%。如果新版本集群处理出现问题，那么要自动降级到老版本集群继续服务。若新版本出现大面积故障，则要将所有流量引入到老版本集群。因此，接入层要能灵活控制流量方向。示意图如下图所示。



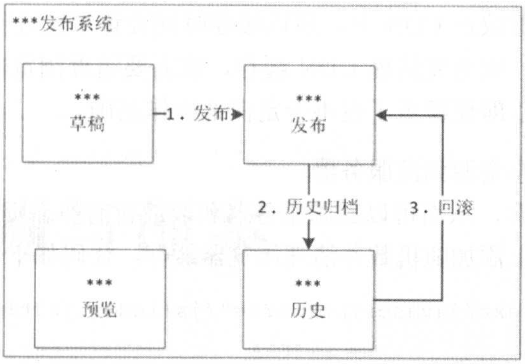
失败降级我们可以借助 Nginx 的 error_page。

```
proxy_intercept_errors on;
recursive_error_pages on;

location ~* "^/(\d+)\.html$" {
    proxy_pass http://new_version/$1.html;
    error_page 500 502 503 504 =200 /fallback_version/$1.html;
}
```

失败降级是很重要的特性，关键时候不至于让用户不能访问或者看到白屏，如果有 CDN，则切换版本时一定要记得去掉 CDN。

7.4 数据版本回滚

有些特定行业业务数据中的商品/价格数据需要进行版本化处理，一方面为了审计需要，另一方面为了出现问题时能及时回滚。版本化设计可以基于下图的架构。

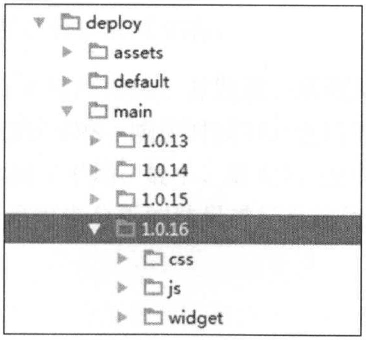


设计版本化数据结构时，有两种思路：全量和增量。全量版本化是指即使只变更了其中一个字段也将整体记录进行历史版本化，保存的数据量比较多，但是回滚方便。而增量版本化是指只保存变化的字段，保存的数据量较少，但是回滚起来很麻烦，需要回溯。因此，为了简单化处理一般采用全量版本化机制。

另外，在设计消息队列时，重要业务会对消息进行副本处理，以便万一业务逻辑出现问题能进行历史数据回滚，从而修复问题。

7.5 静态资源版本回滚

在前端开发中，静态资源版本也是会经常变更的，如 JS/CSS，而每次内容变更时我们都会生成一个全量新版本放到项目的 deploy 目录中，从而保证版本可追溯，出现问题时能及时回滚。目录结构如下图所示。



因为静态资源一般放在 CDN 上，所以缓存时间设置得比较长，比如 1 个月。这样若发布的版本有问题，则需要清理 CDN 缓存，也需要清理浏览器缓存，而且因为存在版本覆盖的问题，所以即使覆盖了也不一定保证操作正确。

- 发布新的静态资源到源服务器。
- 清理CDN缓存，从而可以回源服务器获取最新的静态资源。
- 在新的URL上添加随机数并清理浏览器缓存，代码如下。

```
<script type="text/javascript" src="/js/index.js?time=201610231111"></
script>。
```

而全量版本机制是最可靠的方式，我们先部署全量版本，然后通过如下方式引用。

```
<script type="text/javascript" src="/1.0.16/js/index.js"></script>
```

在当前发布版本出现问题时，只需要将版本号更改为上一个版本号即可，不需要清理 CDN、不需要清理浏览器缓存。

当然，这里要设置合理的服务器端页面缓存时间，比如 2 分钟，用户看到错误的发布版本最多 2 分钟时间。为了方便测试，可以在请求参数中加入版本号，如 http://item.jd.com/2381431.html?version=1.0.15，方便验证老版本或者测试新版本，使得测试或验证多个版本时，不需要来回修改服务器端代码。

8 压测与预案

在大促来临之前，研发人员需要对现有系统进行梳理，发现系统瓶颈和问题，然后进行系统调优来提升系统的健壮性和处理能力。一般通过系统压测来发现系统瓶颈和问题，然后进行系统优化和容灾（如系统参数调优、单机房容灾、多机房容灾等）。即使已经把系统优化和容灾做得非常好了，但也存在一些不稳定因素，如网络、依赖服务的SLA不稳定等，这就需要我们制定应急预案，在出现这些因素后进行路由切换或降级处理。在大促之前需要进行预案演习，确保预案的有效性。

8.1 系统压测

压测一般指性能压力测试，用来评估系统的稳定性和性能，通过压测数据进行系统容量评估，从而决定是否需要进行扩容或缩容。

压测之前要有压测方案〔如压测接口、并发量、压测策略（突发、逐步加压、并发量）、压测指标（机器负载、QPS/TPS、响应时间）〕，之后要产出压测报告〔压测方案、机器负载、QPS/TPS、响应时间（平均、最小、最大）、成功率、相关参数（JVM参数、压缩参数）等〕，最后根据压测报告分析的结果进行系统优化和容灾。

8.1.1 线下压测

通过如 JMeter、Apache ab 压测系统的某个接口（如查询库存接口）或者某个组件（如数据库连接池），然后进行调优（如调整 JVM 参数、优化代码），实现单个接口或组件的性能最优。

线下压测的环境（比如，服务器、网络、数据量等）和线上的完全不一样，仿真度不高，很难进行全链路压测，适合组件级的压测，数据只能作为参考。

8.1.2 线上压测

线上压测的方式非常多，按读写分为读压测、写压测和混合压测，按数据仿真度分为仿真压测和引流压测，按是否给用户提供服务分为隔离集群压测和线上集群压测。

读压测是压测系统的读流量，比如，压测商品价格服务。写压测是压测系统的写流量，比如下单。写压测时，要注意把压测写的数据和真实数据分离，在压测完成后，删除压测数据。只进行读或写压测有时是不能发现系统瓶颈的，因为有时读和写是会相互影响的，因此，这种情况下要进行混合压测。

仿真压测是通过模拟请求进行系统压测，模拟请求的数据可以是使用程序构造、人工构造（如提前准备一些用户和商品），或者使用 Nginx 访问日志，如果压测的数据量有限，则会形成请求热点。而更好的方式可以考虑引流压测，比如使用 TCPCopy 复制线上真实流量，然后引流到压测集群进行压测，还可以将流量放大 N 倍，来观察服务器负载能力。

隔离集群压测是指将对外提供服务的部分服务器从线上集群摘除，然后将线上流量引流到该集群进行压测，这种方式很安全。有时也可以直接对线上集群进行压测，如通过缩减线上服务器数量实现，通过增大单台服务器的负载进行压测，这种方式风险很大，通过逐步减少服务器进行，并且在后半夜用户少的时候进行。

单机压测是指对集群中的一台机器进行压测，从而评估出单机极限处理能力，发现单机的瓶颈点，这样可以把单机性能优化到极致。但实际集群的瓶颈往往是其依赖的系统或服务，如数据库、缓存或者调用的服务，因此单机压测的结果不能反映集群整体处理能力，也需要进行集群压测，从而评估出集群的极限处理能力，从而有针对性地对集群依赖的系统或服务进行优化。

在压测时，也应该选择离散压测，即选择的数据应该是分散的或者长尾的，比如，刚刚新增的商品一般在缓存中，而去年已经下架的商品已经从缓存中移除，刚刚新增的商品往往是热点数据，而去年已下架的商品是冷数据，所以如果压测的数据是热点数据

则是不能反映出系统的真实处理能力。

另外，在实际压测时应该进行全链路压测，因为可能存在一个非核心系统服务调用问题造成整个交易链路出现问题，或者链路中的各个系统存在竞争资源的情况，因此为了保证压测的真实性，应该进行全链路压测，通过全链路压测来发现问题。

8.2 系统优化和容灾

拿到压测报告后，接下来会分析报告，然后进行一些有针对性的优化，如硬件升级、系统扩容、参数调优、代码优化（如代码同步改异步）、架构优化（如加缓存、读写分离、历史数据归档）等。不要把别人的经验或案例拿来直接套在自己的场景下，一定要压测，相信压测数据而不是别人的案例。

在进行系统优化时，要进行代码走查，发现不合理的参数配置，如超时时间、降级策略、缓存时间等。在系统压测中进行慢查询排查，包括 Redis、MySQL 等，通过优化查询解决慢查询问题。系统优化和高并发系统的稳定性保障可扫二维码参考肖飞的《高性能高并发系统的稳定性保障》。

在应用系统扩容方面，可以根据去年流量、与运营业务方沟通促销力度、最近一段时间的流量来评估出是否需要进行扩容，需要扩容多少倍，比如，预计 GMV 增长 100%，那么可以考虑扩容 2~3 倍容量。还要根据系统特点进行评估，如商品详情页可能要支持平常的十几倍流量，如秒杀系统可能要支持平常的几十倍流量。扩容之后还要预留一些机器应对突发情况，在扩容上尽量支持快速扩容，从而出现突发情况时可以几分钟内完成扩容。

不要把所有鸡蛋放进一个篮子，在扩容时要考虑系统容灾，比如分组部署、跨机房部署。容灾是通过部署多组（单机房/多机房）相同应用系统，当其中一组出现问题时，可以切换到另一个分组，保证系统可用。

8.3 应急预案

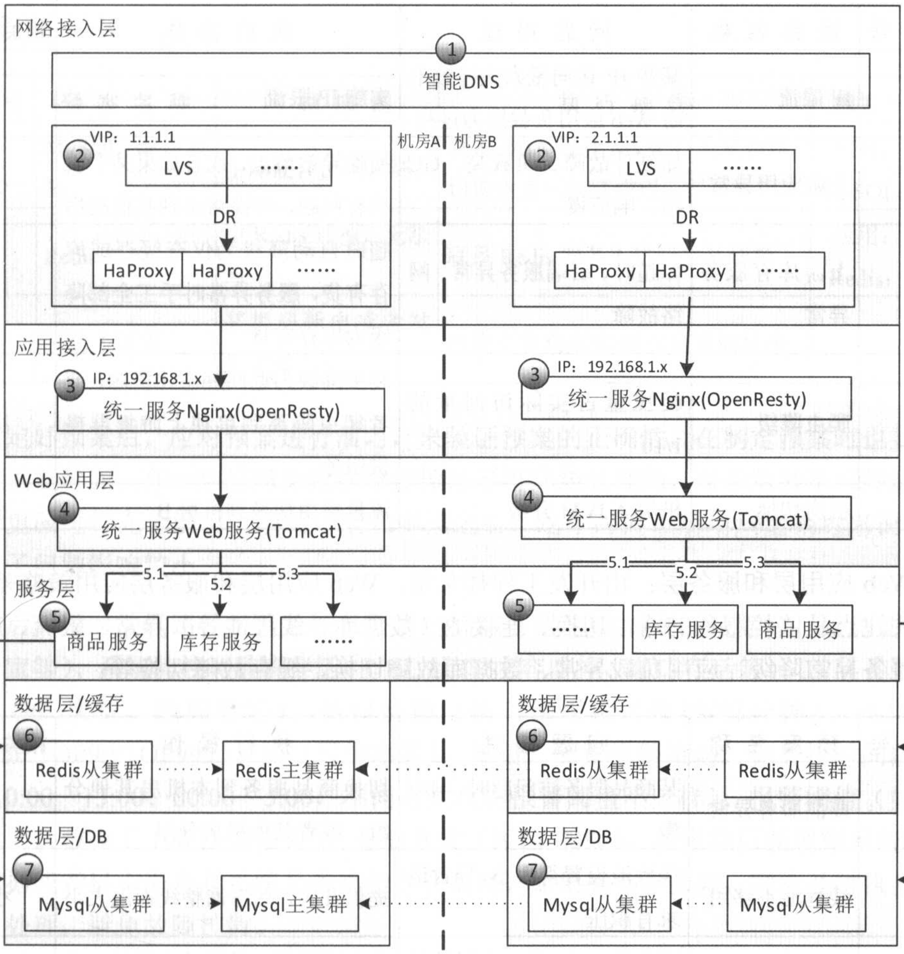
在系统压测之后会发现一些系统瓶颈，在系统优化之后会提升系统吞吐量并降低响应时间，容灾之后的系统可用性得以保障，但还是会存在一些风险，如网络抖动、某台机器负载过高、某个服务变慢、数据库 Load 值过高等，为了防止因为这些问题而出现系统雪崩，需要针对这些情况制定应急预案，从而在出现突发情况时，有相应的措施来解决掉这些问题。

应急预案可按照如下几步进行：首先进行系统分级，然后进行全链路分析、配置监控报警，最后制定应急预案。

系统分级可以按照交易核心系统和交易支撑系统进行划分。交易核心系统，如购物车，如果挂了，将影响用户无法购物，因此需要投入更多资源保障系统质量，将系统优化到极致，降低事故率。而交易支撑系统是外围系统，如商品后台，即使挂了也不影响前台用户购物，这些系统允许暂时不可用。实际系统分级要根据公司特色进行，目的是对不同级别的系统实施不同的质量保障，核心系统要投入更多资源保障系统高可用，外围系统要投入较少资源允许系统暂时不可用。

系统分级后，接下来要对交易核心系统进行全链路分析，从用户入口到后端存储，梳理出各个关键路径，对相关路径进行评估并制定预案。即当出现问题时，该路径可以执行什么操作来保证用户可下单、可购物，并且也要防止问题的级联效应和雪崩效应。

如下图所示，梳理系统全链路关键路径，包括网络接入层、应用接入层、Web 应用层、服务层、数据层等，最后可以按照如下表格制定应急预案。



网络接入层：由系统工程师负责，主要关注是机房不可用、DNS 故障、VIP 故障等预案处理。

序　号	预 案 名 称	问 题 描 述	执 行 操 作	相关干系人
1	机房故障	机房 A 公网入口挂掉	从 DNS 摘除该机房入口，或者切换 DNS 到其他机房	DNS：小 A
2	VIP 网络异常	如某 VIP 出现网络抖动，暂时无法解决	切换到备用 VIP	VIP：小 B

应用接入层：由开发工程师负责，主要关注点是上游应用路由切换、限流、降级、隔离等预案处理。

序号	预案名称	问题描述	执行操作	相关干系人
3	IP 限流	某些 IP 访问量太大导致上游 Web 应用负载压力过高	实施 IP 限流	小 C
3	上游应用异常	如硬件故障、负载高、GC 慢、响应慢	摘除异常节点，或者如果某个机房有问题，切换路由到其他机房	小 C
3	上游库存服务异常	如超时、后端服务异常、网络故障	超时自动降级为仅查缓存或库存有货，服务异常时手工全部降级为库存有货	小 C
3	爬虫降级	爬虫量占实际访问量的 1/10	爬虫降级为返回静态化资源，或者爬虫隔离到单独上游集群提供服务	小 C
3	机房切换	机房 A JVM 升级	流量路由切换到机房 B	小 C

Web 应用层和服务层：由开发工程师负责，Web 应用层和服务层应用策略差不多，主要关注点是依赖服务的路由切换、连接池（数据库、线程池等）异常、限流、超时降级、服务异常降级、应用负载异常、数据库故障切换、缓存故障切换等。

序号	预案名称	问题描述	执行操作	相关干系人
4	商品服务异常	某商品服务访问超时，响应慢	切换商品服务到本机房其他分组，或者其他机房分组	小 D
4	线程池不够用	线程池设置得太小，导致请求有积压	提供开关，动态调整线程池大小	小 D
4	请求量太大	请求量太大，超过实际的处理能力	客户端限流，设置请求阈值，当超出阈值后，请求自动降级处理	小 D
4	商品服务压力过大	由于异常，导致查询商品服务的调用量太大，扛不住	客户端暂停查询，或限流	应用：小 D 商品服务：小 E
4	网络异常	导致服务调用大面积超时	自动切换到其他分组，或者切换到其他机房	小 D
5	调用量太大	商品服务总调用量太大	服务器端限流，设置服务器端总的调用量阈值	小 D

数据层：由开发工程师或系统工程师负责，主要关注点是数据库/缓存负载高、数

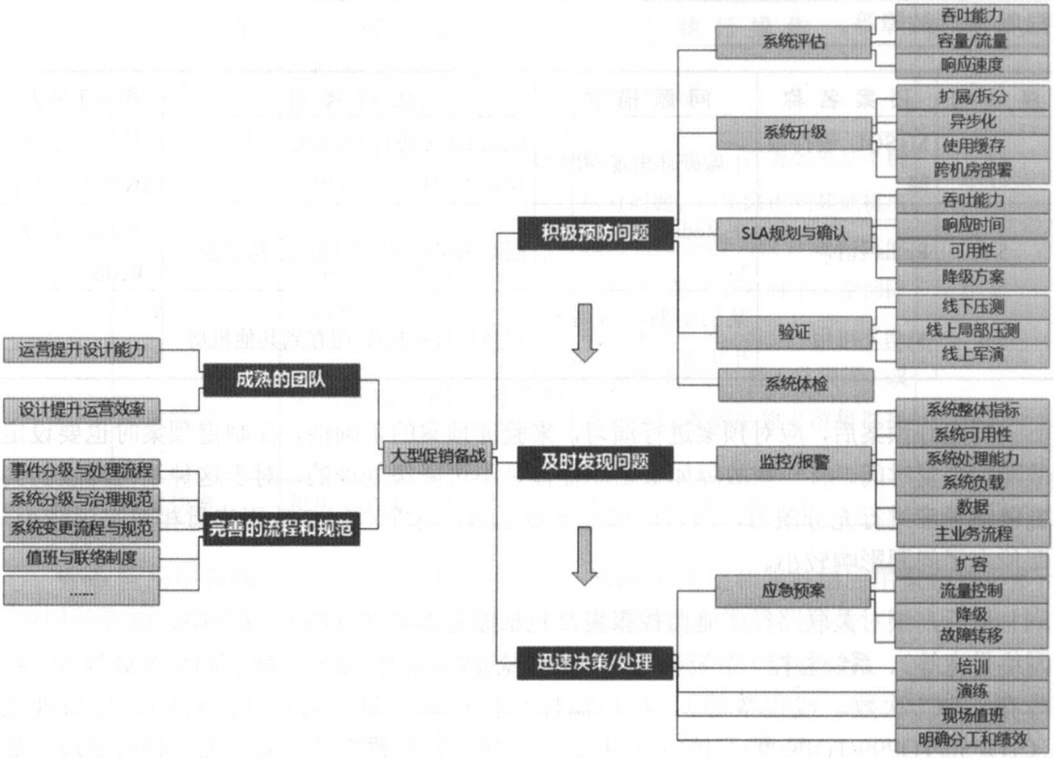
据库/缓存故障等。

序　号	预案名称	问题描述	执行操作	相关干系人
5/6	MySQL 硬件故障	主库硬件出现问题	联系 DBA 进行 MySQL 主从切换，切换完成后验证应用	应用：小 D MySQL：小 F
5/6	Redis 挂掉	主 Redis 挂掉，无法写	联系 Redis 服务人进行主从切换	应用：小 D Redis：小 G
5/6	机房不可用	某机房断电或者发生灾难	切换所有数据库/缓存到其他机房	

制定好预案后，应对预案进行演习，来验证预案的正确性，在制定预案时也要设定故障的恢复时间。有一些故障如数据库挂掉是不可降级处理的，对于这种不可降级的关键链路更应进行充分演习。演习一般在零点之后，这个时间点后用户量相对来说较少，即使出了问题影响较小。

最后，要对关联路径实施监控报警，包括服务器监控（CPU 使用率、磁盘使用率、网络带宽等）、系统监控（系统存活、URL 状态/内容监控、端口存活等）、JVM 监控（堆内存、GC 次数、线程数等）、接口监控〔接口调用量（每秒/每分钟）、接口性能（TOP50/TOP99/TOP999）、接口可用率等〕。然后，配置报警策略，如监控时间段（如上午 10:00—13:00、00:00—5:00，不同时间段的报警阈值不一样）、报警阈值（如每 5 分钟调用次数少于 100 次则报警）、通知方式（短信/邮件）。在报警后要观察系统状态、监控数据或者日志来查看系统是否真的存在故障，如果确实是故障，则应及时执行相关的预案处理，避免故障扩散。

想要了解更多大促备战思路和方法，可扫二维码参考林世洪写的《京东大促备战思路和方法 2.0 解密》。


大型促销备战
成熟的团队
运营提升设计能力
设计提升运营效率
完善的流程和规范
事件分级与处理流程
系统分级与治理规范
系统变更流程与规范
值班与联络制度
......
积极预防问题
系统评估
吞吐能力
容量/流量
响应速度
系统升级
扩展/拆分
异步化
使用缓存
跨机房部署
SLA规划与确认
吞吐能力
响应时间
可用性
降级方案
验证
线下压测
线上局部压测
线上军演
系统体检
及时发现问题
监控/报警
系统整体指标
系统可用性
系统处理能力
系统负载
数据
主业务流程
迅速决策/处理
应急预案
扩容
流量控制
降级
故障转移
培训
演练
现场值班
明确分工和绩效

第 3 部分　高并发

- 应用级缓存
- HTTP 缓存
- 多级缓存
- 连接池线程池详解
- 异步并发实战
- 如何扩容
- 队列术

9 应用级缓存

9.1 缓存简介

缓存，笔者的理解是让数据更接近于使用者，目的是让访问速度更快。工作机制是先从缓存中读取数据，如果没有，再从慢速设备上读取实际数据并同步到缓存。那些经常读取的数据、频繁访问的数据、热点数据、I/O 瓶颈数据、计算昂贵的数据、符合 5 分钟法则和局部性原理的数据都可以进行缓存。如 CPU→L1/L2/L3→内存→磁盘就是一个典型的例子，CPU 需要数据时先从 L1 读取，如果没有找到，则查找 L2/L3 读取，如果没有，则到内存中查找，如果还没有，会到磁盘中查找。还有比如用过 Maven 的读者都应该知道，加载依赖的时候，先从本机仓库找，再从本地服务器仓库找，最后到远程仓库服务器找。另外还有京东的物流为什么那么快？他们在各地都有分仓库，如果该仓库有货物，那么送货的速度是非常快的。

本文以 Java 应用缓存为示例进行讲解。

9.2 缓存命中率

缓存命中率是从缓存中读取数据的次数与总读取次数的比率，命中率越高越好。缓存命中率 = 从缓存中读取次数/〔总读取次数（从缓存中读取次数 + 从慢速设备上读取次数)〕。这是一个非常重要的监控指标，如果做缓存，则应通过监控这个指标来看缓存是否工作良好。

9.3 缓存回收策略

1. 基于空间

基于空间指缓存设置了存储空间，如设置为 10MB，当达到存储空间上限时，按照一定的策略移除数据。

2. 基于容量

基于容量指缓存设置了最大大小，当缓存的条目超过最大大小时，按照一定的策略移除旧数据。

3. 基于时间

TTL（Time To Live）：存活期，即缓存数据从创建开始直到到期的一个时间段（不管在这个时间段内有没有被访问，缓存数据都将过期）。

TTI（Time To Idle）：空闲期，即缓存数据多久没被访问后移除缓存的时间。

4. 基于 Java 对象引用

软引用：如果一个对象是软引用，那么当 JVM 堆内存不足时，垃圾回收器可以回

收这些对象。软引用适合用来做缓存，从而当 JVM 堆内存不足时，可以回收这些对象腾出一些空间供强引用对象使用，从而避免 OOM。

弱引用：当垃圾回收器回收内存时，如果发现弱引用，则将它立即回收。相对于软引用，弱引用有更短的生命周期。

注意：只有在没有其他强引用对象引用弱引用/软引用对象时，垃圾回收时才回收该引用。即如果有一个对象（不是弱引用/软引用对象）引用了弱引用/软引用对象，那么垃圾回收时不会回收该弱引用/软引用对象。

5. 回收算法

使用基于空间和基于容量的缓存会使用一定的策略移除旧数据，常见的如下。

FIFO（First In First Out）：先进先出算法，即先放入缓存的先被移除。

LRU（Least Recently Used）：最近最少使用算法，使用时间距离现在最久的那个被移除。

LFU（Least Frequently Used）：最不常用算法，一定时间段内使用次数（频率）最少的那个被移除。

实际应用中基于 LRU 的缓存居多，如 Guava Cache、Ehcache 支持 LRU。

9.4 Java 缓存类型

堆缓存：使用 Java 堆内存来存储缓存对象。使用堆缓存的好处是没有序列化/反序列化，是最快的缓存。缺点也很明显，当缓存的数据量很大时，GC（垃圾回收）暂停时间会变长，存储容量受限于堆空间大小。一般通过软引用/弱引用来存储缓存对象，即当堆内存不足时，可以强制回收这部分内存释放堆内存空间。一般使用堆缓存存储较热的数据。可以使用 Guava Cache、Ehcache 3.x、MapDB 实现。

堆外缓存：即缓存数据存储在堆外内存，可以减少 GC 暂停时间（堆对象转移到堆外，GC 扫描和移动的对象变少了），可以支持更大的缓存空间（只受机器内存大小限制，不受堆空间的影响）。但是，读取数据时需要序列化/反序列化，因此会比堆缓存慢很多。

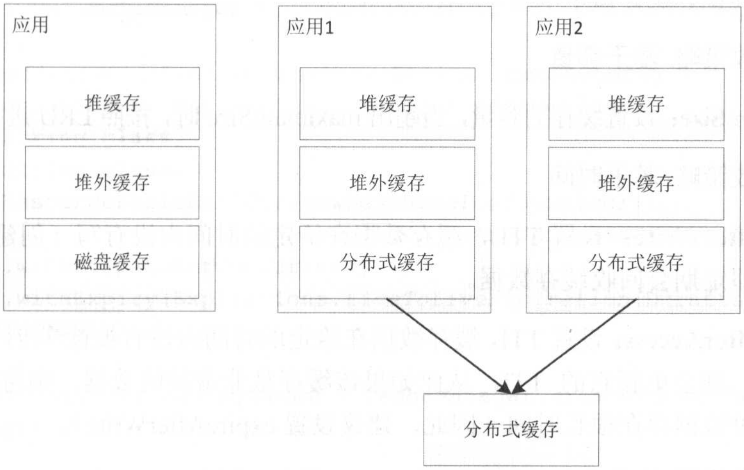
可以使用 Ehcache 3.x、MapDB 实现。

磁盘缓存： 即缓存数据存储在磁盘上，在 JVM 重启时数据还是存在的，而堆缓存/堆外缓存数据会丢失，需要重新加载。可以使用 Ehcache 3.x、MapDB 实现。

分布式缓存： 上文提到的缓存是进程内缓存和磁盘缓存，在多 JVM 实例的情况下，会存在两个问题：1. 单机容量问题；2. 数据一致性问题（多台 JVM 实例的缓存数据不一致怎么办？），不过，这个问题不用太纠结，既然数据允许缓存，则表示允许一定时间内的不一致，因此可以设置缓存数据的过期时间来定期更新数据；3. 缓存不命中时，需要回源到 DB/服务请求多变问题：每个实例在缓存不命中的情况下都会回源到 DB 加载数据，因此，多实例后 DB 整体的访问量就变多了，解决办法是可以使用如一致性哈希分片算法。因此，这些情况可以考虑使用分布式缓存来解决。可以使用 ehcache-clustered（配合 Terracotta server）实现 Java 进程间分布式缓存。当然也可以使用如 Redis 实现分布式缓存。

两种模式如下。

- **单机时：** 存储最热的数据到堆缓存，相对热的数据到堆外缓存，不热的数据到磁盘缓存。
- **集群时：** 存储最热的数据到堆缓存，相对热的数据到堆外缓存，全量数据到分布式缓存。



接下来，我们看看如何在 Java 中使用堆缓存、堆外缓存、磁盘缓存、分布式缓存，是不是感觉像 L1、L2、L3 级缓存架构。

Guava Cache 只提供堆缓存，小巧灵活，性能最好，如果只使用堆缓存，那么使用它就够了。

Ehcache 3.x 提供了堆缓存、堆外缓存、磁盘缓存、分布式缓存。但是，其代码注释比较少，API 还不完善（比如，2.x 支持 LRU、LFU、FIFO，而 3.x 目前还没有 API 设置），功能还不完善（比如，集群情况下，个人测试结果是暂时不可以在生产环境使用），如果需要较稳定的 API 和功能，则请考虑使用 Ehcache 2.x（不支持堆外缓存）。

MapDB 是一款嵌入式 Java 数据库引擎和集合框架。提供了 Maps、Sets、Lists、Queues、Bitmaps 的支持，还支持 ACID 事务、增量备份。支持堆缓存、堆外缓存、磁盘缓存。

9.4.1 堆缓存

1. Gauva Cache 实现

```
Cache<String, String> myCache =
        CacheBuilder.newBuilder()
                .concurrencyLevel(4)
                .expireAfterWrite(10, TimeUnit.SECONDS)
                .maximumSize(10000)
                .build();
```

然后可以通过 put、getIfPresent 来读写缓存。CacheBuilder 有几类参数：缓存回收策略、并发设置、统计命中率等。

缓存回收策略/基于容量

maximumSize：设置缓存的容量，当超出 maximumSize 时，按照 LRU 进行缓存回收。

缓存回收策略/基于时间

expireAfterWrite：设置 TTL，缓存数据在给定的时间内没有写（创建/覆盖）时，则被回收，即定期会回收缓存数据。

expireAfterAccess：设置 TTI，缓存数据在给定的时间内没有被读/写时，则被回收。每次访问时，都会更新它的 TTI，从而如果该缓存是非常热的数据，则将一直不过期，可能会导致脏数据存在很长时间（因此，建议设置 expireAfterWrite）。

缓存回收策略/基于 Java 对象引用

weakKeys/weakValues：设置弱引用缓存。

softValues：设置软引用缓存。

缓存回收策略/主动失效

invalidate(Object key)/ invalidateAll(Iterable<?> keys)/invalidateAll()：主动失效某些缓存数据。

什么时候触发失效呢？Guava Cache 不会在缓存数据失效时立即触发回收操作（如果要这么做，则需要有额外的线程来进行清理），而在 PUT 时会主动进行一次缓存清理，当然读者也可以根据实际业务通过自己设计线程来调用 cleanUp 方法进行清理。

并发级别

concurrencyLevel：Guava Cache 重写了 ConcurrentHashMap，concurrencyLevel 用来设置 Segment 数量，concurrencyLevel 越大并发能力越强。

统计命中率

recordStats：启动记录统计信息，比如命中率等。

2. Ehcache 3.x 实现

本文使用最新的 Ehcache 3.1.2，目前 Ehcache 3.x 版本还比较新，一些文档还不是很完善。

```
CacheManager cacheManager = CacheManagerBuilder. newCacheManagerBuilder().
build(true);
CacheConfigurationBuilder<String, String> cacheConfig =
CacheConfigurationBuilder.newCacheConfigurationBuilder(
        String.class,
        String.class,
        ResourcePoolsBuilder.newResourcePoolsBuilder()
                            .heap(100, EntryUnit.ENTRIES))
        .withDispatcherConcurrency(4)
        .withExpiry(Expirations.timeToLiveExpiration(Duration.of(10,
TimeUnit.SECONDS)));

Cache<String, String> myCache = cacheManager.createCache("myCache",
cacheConfig);
```

CacheManager 在 JVM 关闭时调用 CacheManager.close()方法，可以通过 PUT、GET 来读写缓存。CacheConfigurationBuilder 也有几类参数：缓存回收策略、并发设置、统计命中率等。

缓存回收策略/基于容量

heap(100, EntryUnit.ENTRIES)：设置缓存的条目数量，当超出此数量时按照 LRU 进行缓存回收。

缓存回收策略/基于空间

heap(100, MemoryUnit.MB)：设置缓存的内存空间，当超出此空间时按照 LRU 进行缓存回收。另外，应该设置 withSizeOfMaxObjectGraph(2)统计对象大小时对象图遍历深度和 withSizeOfMaxObjectSize(1, MemoryUnit.*KB*)可缓存的最大对象大小。

缓存回收策略/基于时间

withExpiry(Expirations.*timeToLiveExpiration*(Duration.*of*(10, TimeUnit.*SECONDS*)))：设置 TTL，没有 TTI。

withExpiry(Expirations.*timeToIdleExpiration*(Duration.*of*(10, TimeUnit.*SECONDS*)))：同时设置 TTL 和 TTI，且 TTL 和 TTI 值一样。

缓存回收策略/主动失效

remove(K key)/ removeAll(Set<? extends K> keys)/clear()：主动失效某些缓存数据。

什么时候触发失效呢？Ehcache 使用了类似于 Guava Cache 的机制。

并发级别

目前还没有提供 API 来设置，Ehcache 内部使用 ConcurrentHashMap 作为缓存存储，默认并发级别 16。withDispatcherConcurrency 是用来设置事件分发时的并发级别。

统计命中率

目前还没有开放 API 来统计。

3. MapDB 3.x 实现

```
HTreeMap myCache =
        DBMaker.heapDB().concurrencyScale(16).make().hashMap("myCache")
        .expireMaxSize(10000)
        .expireAfterCreate(10, TimeUnit.SECONDS)
        .expireAfterUpdate(10, TimeUnit.SECONDS)
        .expireAfterGet(10, TimeUnit.SECONDS)
```

```
        .create();
```

然后可以通过 PUT、GET 来读写缓存。其有几类参数：缓存回收策略、并发设置、统计命中率等。

缓存回收策略/基于容量

expireMaxSize： 设置缓存的容量，当超出 expireMaxSize 时，按照 LRU 进行缓存回收。

缓存回收策略/基于时间

expireAfterCreate/expireAfterUpdate： 设置 TTL，缓存数据在给定的时间内没有写（创建/覆盖）时，则被回收，即定期地会回收缓存数据。

expireAfterGet： 设置 TTI，缓存数据在给定的时间内没有被读/写时，则被回收。每次访问时都会更新它的 TTI，从而如果该缓存是非常热的数据，则将一直不过期，可能会导致脏数据存在很长的时间（因此，建议要设置 expireAfterCreate/expireAfterUpdate）。

缓存回收策略/主动失效

remove(Object key) /clear()： 主动失效某些缓存数据。

什么时候触发失效呢？MapDB 默认使用类似于 Guava Cache 的机制。不过，也支持通过如下配置使用线程池定期进行缓存失效。

```
.expireExecutor(scheduledExecutorService)
.expireExecutorPeriod(3000)
```

并发级别

concurrencyScale： 类似于 Guava Cache 的配置。

统计命中率

暂无。

还可以使用 DBMaker.memoryDB()创建堆缓存，它将数据序列化并存储到 1MB 大小的 byte[]数组中，从而减少垃圾回收的影响。

9.4.2 堆外缓存

1. EhCache 3.x 实现

```
CacheConfigurationBuilder<String, String> cacheConfig =
```

```
CacheConfigurationBuilder.newCacheConfigurationBuilder(
          String.class,
          String.class,
          ResourcePoolsBuilder.newResourcePoolsBuilder()
                    .offheap(100, MemoryUnit.MB))
          .withDispatcherConcurrency(4)
          .withExpiry(Expirations.timeToLiveExpiration(Duration.of(10,
TimeUnit.SECONDS)))
          .withSizeOfMaxObjectGraph(3)
          .withSizeOfMaxObjectSize(1, MemoryUnit.KB);
```

堆外缓存不支持基于容量的缓存过期策略。

2. MapDB 3.x 实现

```
HTreeMap myCache =

DBMaker.memoryDirectDB().concurrencyScale(16).make().hashMap("myCache")
          .expireStoreSize(64 * 1024 * 1024) //指定堆外缓存大小 64MB
          .expireMaxSize(10000)
          .expireAfterCreate(10, TimeUnit.SECONDS)
          .expireAfterUpdate(10, TimeUnit.SECONDS)
          .expireAfterGet(10, TimeUnit.SECONDS)
          .create();
```

在使用堆外缓存时，请记得添加 JVM 启动参数，如-XX:MaxDirectMemorySize=10G。

9.4.3 磁盘缓存

1. EhCache 3.x 实现

```
CacheManager cacheManager = CacheManagerBuilder. newCacheManagerBuilder()
          //默认线程池
          .using(PooledExecutionServiceConfigurationBuilder
                    .newPooledExecutionServiceConfigurationBuilder()
                    .defaultPool("default", 1, 10).build())
          //磁盘文件存储位置
         .with(new CacheManagerPersistenceConfiguration(new File("D:\\bak")))
          .build(true);

CacheConfigurationBuilder<String, String> cacheConfig =
    CacheConfigurationBuilder. newCacheConfigurationBuilder(
          String.class,
```

```
        String.class,
        ResourcePoolsBuilder.newResourcePoolsBuilder()
                            .disk(100, MemoryUnit.MB, true)) //磁盘缓存
        .withDiskStoreThreadPool("default", 5)
//使用"default"线程池进行 dump 文件到磁盘
        .withExpiry(Expirations.timeToLiveExpiration(Duration.of(50,
TimeUnit.SECONDS)))
        .withSizeOfMaxObjectGraph(3)
        .withSizeOfMaxObjectSize(1, MemoryUnit.KB);
```

在 JVM 停止时，记得调用 **cacheManager**.close()，从而保证内存数据能 dump 到磁盘。

2. MapDB 3.x 实现

```
DB db = DBMaker
        .fileDB("D:\\bak\\a.data") //数据存哪里
        .fileMmapEnable() //启用 mmap
        .fileMmapEnableIfSupported() //在支持的平台上启用 mmap
        .fileMmapPreclearDisable() //让 mmap 文件更快
        .cleanerHackEnable() //一些 BUG 处理
        .transactionEnable() //启用事务
        .closeOnJvmShutdown()
        .concurrencyScale(16)
        .make();

HTreeMap myCache = db.hashMap("myCache")
        .expireMaxSize(10000)
        .expireAfterCreate(10, TimeUnit.SECONDS)
        .expireAfterUpdate(10, TimeUnit.SECONDS)
        .expireAfterGet(10, TimeUnit.SECONDS)
        .createOrOpen();
```

因为开启了事务，MapDB 则开启了 WAL。另外，操作完缓存后记得调用 db.commit 方法提交事务。

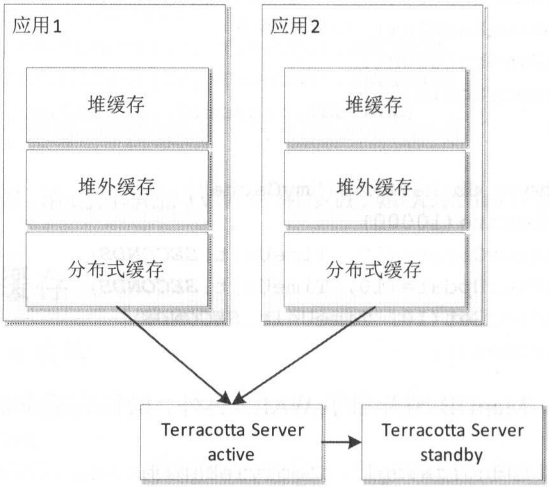
```
myCache.put("key" + counterWriter, "value" + counterWriter);
db.commit();
```

9.4.4 分布式缓存

本文使用 Ehcache 3.1+Terracotta server 实现，Ehcache 3.1 引入了一个下载套件，其包含了 Terracotta Server。

调用 start-tc-server 脚本启动 tc server。

1. 架构



2. Terracotta Server 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<tc-config xmlns="http://www.terracotta.org/config"

xmlns:ohr="http://www.terracotta.org/config/offheap-resource">
```

```
<servers>
  <server host="192.168.147.50" name="s1">
   <tsa-port>9510</tsa-port>
   <tsa-group-port>9530</tsa-group-port>
  </server>

  <server host="192.168.147.52" name="s2">
   <tsa-port>9510</tsa-port>
   <tsa-group-port>9530</tsa-group-port>
  </server>

 <client-reconnect-window>30</client-reconnect-window>
 <restartable enabled="true"/>
 </servers>

 <services>
     <service id="resources">
        <ohr:offheap-resources>
           <ohr:resource name="cache" unit="MB">64</ohr:resource>
        </ohr:offheap-resources>
     </service>
  </services>
</tc-config>
```

配置了两个 tc server，一主一备。在两台服务器中分别调用如下脚本启动两台 tc server。

```
./start-tc-server.sh -f tc-config.xml  -n s1
./start-tc-server.sh -f tc-config.xml  -n s2
```

3. Ehcache 代码片段

```
CacheManagerBuilder<PersistentCacheManager>
clusteredCacheManagerBuilder =
        CacheManagerBuilder.newCacheManagerBuilder()
        .with(ClusteringServiceConfigurationBuilder.cluster(URI.create(
"terracotta://192.168.147.50:9510")).readOperationTimeout(500,
TimeUnit.MILLISECONDS).autoCreate());

final PersistentCacheManager cacheManager = clusteredCacheManagerBuilder.
build(true);


Cache<String, String> myCache = cacheManager.createCache("myCache",
      CacheConfigurationBuilder.newCacheConfigurationBuilder(
```

```
                String.class,
                String.class,
                ResourcePoolsBuilder.newResourcePoolsBuilder()
                        .with(ClusteredResourcePoolBuilder
                                .clusteredDedicated(
                                        "cache", 32, MemoryUnit.MB)))
                .withDispatcherConcurrency(4)
                .withExpiry(Expirations.timeToLiveExpiration(Duration.of
(10, TimeUnit.SECONDS))));
```

可以看到一个问题，此处只指定了 IP 为 192.168.147.50 这台机器的 tc server，那么当 50 这台机器挂了时，目前是不能自动连接到 52 机器的。不知道未来 Ehcache 是否会支持，或者可以考虑使用其主打产品 BigMemory（付费）。

对于分布式缓存个人还是喜欢使用 Redis 之类的，性能也非常好，有主从模式、集群模式。目前不建议使用 Ehcache 3.1+Terracotta server 组合。

9.4.5 多级缓存

如先查找堆缓存，如果没有则查找磁盘缓存，那么使用 MapDB 通过如下配置实现。

```
HTreeMap diskCache = db.hashMap("myCache")
        .expireStoreSize(8 * 1024 * 1024 * 1024)
        .expireMaxSize(10000)
        .expireAfterCreate(10, TimeUnit.SECONDS)
        .expireAfterUpdate(10, TimeUnit.SECONDS)
        .expireAfterGet(10, TimeUnit.SECONDS)
        .createOrOpen();


HTreeMap heapCache = db.hashMap("myCache")
        .expireMaxSize(100)
        .expireAfterCreate(10, TimeUnit.SECONDS)
        .expireAfterUpdate(10, TimeUnit.SECONDS)
        .expireAfterGet(10, TimeUnit.SECONDS)
        .expireOverflow(diskCache) //当缓存溢出时存储到disk
        .createOrOpen();
```

对于更复杂的多级缓存请参考“第 11 章 多级缓存”。

9.5 应用级缓存示例

9.5.1 多级缓存 API 封装

我们的业务数据如商品类目、店铺、商品基本信息等都可以进行适当的本地缓存，以提升性能。对于多实例的情况，不仅会使用本地缓存，还会使用分布式缓存，因此需要进行适当的 API 封装以简化缓存操作。

1. 本地缓存初始化

```
public class LocalCacheInitService extends BaseService {
    @Override
    public void afterPropertiesSet() throws Exception {
        //商品类目缓存
        Cache<String, Object> categoryCache =
                CacheBuilder.newBuilder()
                        .softValues()
                        .maximumSize(1000000)
                        .expireAfterWrite(Switches.CATEGORY.getExpiresInSe
conds() / 2, TimeUnit.SECONDS)
                        .build();
        addCache(CacheKeys.CATEGORY_KEY, categoryCache);
    }

    private void addCache(String key, Cache<?, ?> cache) {
        localCacheService.addCache(key, cache);
    }
}
```

本地缓存过期时间使用分布式缓存过期时间的一半，防止本地缓存数据缓存时间太长造成多实例间的数据不一致。

另外，将缓存 key 前缀与本地缓存关联，从而匹配缓存 key 前缀，就可以找到相关联的本地缓存。

2. 写缓存 API 封装

先写本地缓存，如果需要写分布式缓存，则通过异步更新分布式缓存。

```
public void set(final String key, final Object value,
        final int remoteCacheExpiresInSeconds) throws RuntimeException {
    if (value == null) {
        return;
    }

    //复制值对象
    //本地缓存是引用，分布式缓存需要序列化
    //如果不复制的话，则假设数据更改后将造成本地缓存与分布式缓存不一致
    final Object finalValue = copy(value);
    //如果配置了写本地缓存，则根据 KEY 获得相关的本地缓存，然后写入
    if (writeLocalCache) {
        Cache localCache = getLocalCache(key);
        if (localCache != null) {
            localCache.put(key, finalValue);
        }
    }
    //如果配置了不写分布式缓存，则直接返回
    if (!writeRemoteCache) {
        return;
    }
    //异步更新分布式缓存
    asyncTaskExecutor.execute(() -> {
        try {
            redisCache.set(
                    key,
                    JSONUtils.toJSON(finalValue),
                    remoteCacheExpiresInSeconds);
        } catch (Exception e) {
            LOG.error("update redis cache error, key : {}", key, e);
        }
    });
}
```

此处使用了异步更新，目的是尽快返回用户请求。而因为有本地缓存，所以即使分布式缓存更新比较慢又产生了回源，也可以在本地缓存命中。

3. 读缓存 API 封装

先读本地缓存，本地缓存不命中的再批量查询分布式缓存，在查询分布式缓存时通

过分区批量查询。

```
private Map innerMget(List<String> keys, List<Class> types)
                                                throws Exception {
    Map<String, Object> result = Maps.newHashMap();
    List<String> missKeys = Lists.newArrayList();
    List<Class> missTypes = Lists.newArrayList();
    //如果配置了读本地缓存，则先读本地缓存
    if(readLocalCache) {
        for (int i = 0; i < keys.size(); i++) {
            String key = keys.get(i);
            Class type = types.get(i);
            Cache localCache = getLocalCache(key);
            if (localCache != null) {
                Object value = localCache.getIfPresent(key);
                result.put(key, value);
                if (value == null) {
                    missKeys.add(key);
                    missTypes.add(type);
                }
            } else {
                missKeys.add(key);
                missTypes.add(type);
            }
        }
    }
    //如果配置了不读分布式缓存，则返回
    if(!readRemoteCache) {
        return result;
    }
    final Map<String, String> missResult = Maps.newHashMap();

    //对 key 分区，不要一次性批量调用太大
    final List<List<String>> keysPage = Lists.partition(missKeys, 10);
    List<Future<Map<String, String>>> pageFutures = Lists.newArrayList();

    try {
        //批量获取分布式缓存数据
        for(final List<String> partitionKeys : keysPage) {
            pageFutures.add(asyncTaskExecutor.submit(
                        () -> redisCache.mget(partitionKeys)));
        }
        for(Future<Map<String, String>> future : pageFutures) {
            missResult.putAll(future.get(3000, TimeUnit.MILLISECONDS));
```

```
            }
        } catch (Exception e) {
            pageFutures.forEach(future -> future.cancel(true));
            throw e;
        }
        //合并 result 和 missResult，此处实现省略
        return result;
    }
```

此处将批量读缓存进行了分区，防止乱用批量获取 API。

9.5.2 NULL Cache

首先，定义 NULL 对象。

```
private static final String NULL_STRING = new String();
```

当 DB 没有数据时，写入 NULL 对象到缓存。

```
//查询 DB
String value = loadDB();
//如果 DB 没有数据，则将其封装为 NULL_STRING 并放入缓存
if(value == null) {
    value = NULL_STRING;
}
myCache.put(id, value);
```

读取数据时，如果发现 NULL 对象，则返回 null，而不是回源到 DB。

```
value = suitCache.getIfPresent(id);
//DB 没有数据，返回 null
if(value == NULL_STRING) {
    return null;
}
```

通过这种方式可以防止当 key 对应的数据在 DB 中不存在时频繁查询 DB 的情况。

9.5.3 强制获取最新数据

在实际应用中，我们经常需要强制更新数据，此时就不能使用缓存数据了，可以通过配置 ThreadLocal 开关来决定是否强制刷新缓存（refresh 方法要配合 CacheLoader 一起使用）。

```
if(ForceUpdater.isForceUpdateMyInfo()) {
```

```
        myCache.refresh(skuId);
    }
    String result = myCache.get(skuId);
    if(result == NULL_STRING) {
        return null;
    }
```

9.5.4 失败统计

```
private LoadingCache<String, AtomicInteger> failedCache =
        CacheBuilder.newBuilder()
                .softValues()
                .maximumSize(10000)
                .build(new CacheLoader<String, AtomicInteger>() {
                    @Override
                    public AtomicInteger load(String skuId) throws Exception {
                        return new AtomicInteger(0);
                    }
                });
```

当失败时，通过 failedCache.getUnchecked(id).incrementAndGet()增加失败次数；当成功时，使用 failedCache.invalidate(id)使缓存失效。通过这种方式可以控制失败重试次数。而且当内存不足时，缓存数据可以被垃圾回收以腾出一些空间。

9.5.5 延迟报警

```
private static LoadingCache<String, Integer> alarmCache =
        CacheBuilder.newBuilder()
                .softValues()
                .maximumSize(10000).expireAfterAccess(1, TimeUnit.HOURS)
                .build(new CacheLoader<String, Integer>() {
                    @Override
                    public Integer load(String key) throws Exception {
                        return 0;
                    }
                });

//报警代码
Integer count = 0;
if(redis != null) {
    String countStr =
            Objects.firstNonNull(redis.opsForValue().get(key), "0");
    count = Integer.valueOf(countStr);
```

```
} else {
    count = alarmCache.get(key);
}
if(count % 5 == 0) { //5 次报一次
    //报警
}
count = count + 1;
if(redis != null) {
    redis.opsForValue().set(key, String.valueOf(count),
                    1, TimeUnit. HOURS);
} else {
    alarmCache.put(key, count);
}
```

如果一出问题就报警，则存在报警量过多或者假报警的问题，因此，可以考虑 *N* 久报警了 *M* 次，才真正报警。此时，也可以使用 Cache 来统计。本示例还加入了 Redis 分布式缓存记录支持。

9.6 缓存使用模式实践

前面已经介绍了 Java 缓存的使用。对于我们来说，如果有人已总结出一些缓存使用模式/模板，我们在使用时直接照搬模式即可。确实已经有总结好的模式，主要分两大类：Cache-Aside 和 Cache-As-SoR（Read-through、Write-through、Write-behind）。

首先，介绍三个名词。

SoR（system-of-record）： 记录系统，或者可以叫做数据源，即实际存储原始数据的系统。

Cache： 缓存，是 SoR 的快照数据，Cache 的访问速度比 SoR 要快，放入 Cache 的目的是提升访问速度，减少回源到 SoR 的次数。

回源： 即回到数据源头获取数据，Cache 没有命中时，需要从 SoR 读取数据，这叫做回源。

本文主要以 Guava Cache 和 Ehcache3.x 作为实践框架来讲解。

9.6.1 Cache-Aside

Cache-Aside 即业务代码围绕着 Cache 写，是由业务代码直接维护缓存，示例代码如下所示。

```
//1. 先从缓存中获取数据
value = myCache.getIfPresent(key);
if(value == null) {
   //2.1. 如果缓存没有命中，则回源到 SoR 获取源数据
   value = loadFromSoR(key);
   //2.2. 将数据放入缓存，下次即可从缓存中获取数据
   myCache.put(key, value);
}
```

读场景，先从缓存获取数据，如果没有命中，则回源到 SoR 并将源数据放入缓存供下次读取使用。

写场景，先将数据写入 SoR，写入成功后立即将数据同步写入缓存。

```
//1. 先将数据写入 SoR
writeToSoR(key, value);
//2. 执行成功后立即同步写入缓存
myCache.put(key, value);
```

或者先将数据写入 SoR，写入成功后将缓存数据过期，下次读取时再加载缓存。

```
//1. 先将数据写入 SoR
writeToSoR(key, value);
//2. 失效缓存，然后下次读时再加载缓存
myCache.invalidate(key);
```

Cache-Aside 适合使用 AOP 模式去实现，具体操作可以参考笔者的博客《Spring Cache 抽象详解》。

对于 Cache-Aside，可能存在并发更新情况，即如果多个应用实例同时更新，那么缓存怎么办？

- 如果是用户维度的数据（如订单数据、用户数据），这种几率非常小，因为并发的情况很少，可以不考虑这个问题，加上过期时间来解决即可。
- 对于如商品这种基础数据，可以考虑使用canal订阅binlog，来进行增量更新分布式缓存，这样不会存在缓存数据不一致的情况。但是，缓存更新会存在延迟。而本地缓存可根据不一致容忍度设置合理的过期时间。

- 读服务场景，可以考虑使用一致性哈希，将相同的操作负载均衡到同一个实例，从而减少并发几率。或者设置比较短的过期时间，可参考“第17章 京东商品详情页服务闭环实践”。

9.6.2 Cache-As-SoR

Cache-As-SoR 即把 Cache 看作为 SoR，所有操作都是对 Cache 进行，然后 Cache 再委托给 SoR 进行真实的读/写。即业务代码中只看到 Cache 的操作，看不到关于 SoR 相关的代码。有三种实现：read-through、write-through、write-behind。

9.6.3 Read-Through

Read-Through，业务代码首先调用 Cache，如果 Cache 不命中由 Cache 回源到 SoR，而不是业务代码（即由 Cache 读 SoR）。使用 Read-Through 模式，需要配置一个 CacheLoader 组件用来回源到 SoR 加载源数据。Guava Cache 和 Ehcache 3.x 都支持该模式。

1. Guava Cache 实现

```
LoadingCache<Integer, Result<Category>> getCache =
    CacheBuilder.newBuilder()
            .softValues()
            .maximumSize(5000).expireAfterWrite(2, TimeUnit.MINUTES)
            .build(new CacheLoader<Integer, Result<Category>>() {
               @Override
               public Result<Category> load(final Integer sortId)
                                                   throws Exception {
                   return categoryService.get(sortId);
               }
        });
```

在 build Cache 时，传入一个 CacheLoader 用来加载缓存，操作流程如下。

（1）应用业务代码直接调用 **getCache**.get(sortId)。

（2）首先查询 Cache，如果缓存中有，则直接返回缓存数据。

（3）如果缓存没有命中，则委托给 CacheLoader，CacheLoader 会回源到 SoR 查询源数据（返回值必须不为 null，可以包装为 Null 对象），然后写入缓存。

使用 CacheLoader 后有几个好处。

- 应用业务代码更简洁了，不需要像Cache-Aside模式那样缓存查询代码和SoR代码交织在一起。如果缓存使用逻辑散落在多处，则使用这种方式很简单地消除了重复代码。
- 解决Dog-pile effect，即当某个缓存失效时，又有大量相同的请求没命中缓存，从而使请求同时到后端，导致后端压力太大，此时限定一个请求去拿即可。

```
if (firstCreateNewEntry) {//第一个请求加载缓存的线程去 SoR 加载源数据
  try {
    synchronized (e) {
      return loadSync(key, hash, loadingValueReference, loader);
    }
  } finally {
    statsCounter.recordMisses(1);
  }
} else {//其他并发线程等待“第一个线程”加载的数据
  return waitForLoadingValue(e, key, valueReference);
}
```

Guava Cache 还支持 get(K key, Callable<? **extends** V> valueLoader)方法，传入一个 Callable 实例，当缓存没命中时，会调用 Callable#call 来查询 SoR 加载源数据。

2. Ehcache 3.x 实现

```
CacheManager cacheManager =
       CacheManagerBuilder. newCacheManagerBuilder(). build(true);
org.ehcache.Cache<String, String> myCache =
     cacheManager. createCache ("myCache",
          CacheConfigurationBuilder
               .newCacheConfigurationBuilder(
                  String.class, String.class,
                  ResourcePoolsBuilder.newResourcePoolsBuilder()
                                        .heap(100, MemoryUnit.MB))
               .withDispatcherConcurrency(4)
               .withExpiry(Expirations.timeToLiveExpiration(Duration.of
(10, TimeUnit.SECONDS)))
              .withLoaderWriter(
                 new DefaultCacheLoaderWriter<String, String> () {
                    @Override
                    public String load(String key) throws Exception {
                       return readDB(key);
                    }
                    @Override
                    public Map<String, String> loadAll(Iterable<? extends
```

```
String> keys) throws BulkCacheLoadingException, Exception {
                        return null;
                    }
                }));
```

Ehcache 3.x 使用 CacheLoaderWriter 来实现，通过 load(K key)和 loadAll(Iterable<? **extends** K> keys)分别来加载单个 key 和批量 key。Ehcache 3.1 没有自己去解决 Dog-pile effect。

9.6.4 Write-Through

Write-Through，被称为穿透写模式/直写模式——业务代码首先调用 Cache 写（新增/修改）数据，然后由 Cache 负责写缓存和写 SoR，而不是由业务代码。使用 Write-Through 模式需要配置一个 CacheWriter 组件用来回写 SoR。Guava Cache 没有提供支持。Ehcache 3.x 支持该模式。Ehcache 需要配置一个 CacheLoaderWriter，CacheLoaderWriter 知道如何去写 SoR。当 Cache 需要写（新增/修改）数据时，首先调用 CacheLoaderWriter 来（立即）同步到 SoR，成功后会更新缓存。

```
   CacheManager cacheManager =
         CacheManagerBuilder.newCacheManagerBuilder ().build(true);
   org.ehcache.Cache<String, String> myCache =
         cacheManager.createCache ("myCache",
           CacheConfigurationBuilder
               .newCacheConfigurationBuilder(String.class, String.class,
                    ResourcePoolsBuilder.newResourcePoolsBuilder()
                                        .heap(100, MemoryUnit.MB))
              .withDispatcherConcurrency(4)
              .withExpiry(Expirations.timeToLiveExpiration(Duration.of(
10, TimeUnit.SECONDS)))
             .withLoaderWriter(
               new DefaultCacheLoaderWriter<String, String> () {
                 @Override
                 public void write(String key, String value)
                                               throws Exception {
                     //write
                 }
                 @Override
                 public void writeAll(Iterable<? extends Map.Entry<?
extends String, ? extends String>> entries) throws BulkCacheWritingException,
Exception {
                     for(Object entry : entries) {
                        //batch write
```

```
                }
            }
            @Override
            public void delete(String key) throws Exception {
                //delete
            }
            @Override
            public void deleteAll(Iterable<? extends String> keys)
throws BulkCacheWritingException, Exception {
                for(Object key : keys) {
                    //batch delete
                }
            }
        }).build());
```

Ehcache 3.x 还是使用 CacheLoaderWriter 来实现，通过 write(String key, String value)、writeAll(Iterable<? extends Map.Entry<? extends String, ? extends String>> entries)和 delete(String key)、deleteAll(Iterable<? extends String> keys)分别来支持单个写、批量写和单个删除、批量删除操作。操作流程如下。

1．当我们调用 myCache.put("e", "123")或者 myCache.putAll(map)时，写缓存。

2．首先，Cache 会将写操作立即委托给 CacheLoaderWriter#write 和#writeAll，然后由 CacheLoaderWriter 负责立即去写 SoR。

3．当写 SoR 成功后，再写入 Cache。

9.6.5 Write-Behind

Write-Behind，也叫 Write-Back，我们称之为回写模式。不同于 Write-Through 是同步写 SoR 和 Cache，Write-Behind 是异步写。异步之后可以实现批量写、合并写、延时和限流。

1．异步写

```
CacheManager cacheManager = CacheManagerBuilder. newCacheManagerBuilder()
      .using(PooledExecutionServiceConfigurationBuilder
            .newPooledExecutionServiceConfigurationBuilder()
            .pool("writeBehindPool", 1, 5)
            .build())
      .build(true);
org.ehcache.Cache<String, String> myCache =
      cacheManager. createCache ("myCache",
```

```
                CacheConfigurationBuilder
                .newCacheConfigurationBuilder(String.class, String.class,
                        ResourcePoolsBuilder.newResourcePoolsBuilder()
                                        .heap(100, MemoryUnit.MB))
                .withDispatcherConcurrency(4)
                .withExpiry(Expirations.timeToLiveExpiration(Duration.of
(10, TimeUnit.SECONDS)))
                .withLoaderWriter(
                  new DefaultCacheLoaderWriter<String, String >() {
                    @Override
                    public void write(String key, String value)
                                            throws Exception {
                      //write
                    }

                    @Override
                    public void delete(String key) throws Exception {
                      //delete
                    }
                  })
                .add(WriteBehindConfigurationBuilder
                      .newUnBatchedWriteBehindConfiguration()
                      .queueSize(5)
                      .concurrencyLevel(2
                      .useThreadPool("writeBehindPool")
                      .build()));
```

几个重要配置如下。

- **ThreadPool：**使用PooledExecutionServiceConfigurationBuilder配置线程池；然后WriteBehindConfigurationBuilder通过useThreadPool配置使用哪一个线程池。
- **WriteBehindConfigurationBuilder：**配置WriteBehind策略。
- **CacheLoaderWriter：**配置WriteBehind如何操作SoR。

WriteBehindConfigurationBuilder 会进行如下几个配置。

- **newUnBatchedWriteBehindConfiguration：**表示不进行批量处理。如配置，那么所有批量操作都将会转换成单个操作，即CacheLoaderWriter只需要实现write和delete即可。
- **queueSize(int size)：**因为操作是异步回写SoR，需要将操作先放入写操作等待队列。因此，可使用queue size定义写操作等待队列最大大小，即线程池队列大小。内部使用NonBatchingLocalHeapWriteBehindQueue。

- concurrencyLevel(**int** concurrency)：配置使用多少个并发线程和队列进行WriteBehind。因为我们只传入一个线程池，是如何实现该模式的呢？首先看如下代码片段。

```
for (int i = 0; i < writeBehindConcurrency; i++) {
  if (config.getBatchingConfiguration() == null) {
    this.stripes.add(
            new NonBatchingLocalHeapWriteBehindQueue<K, V>(
                    executionService, defaultThreadPool,
                    config, cacheLoaderWriter));
  } else {
    this.stripes.add(
            new BatchingLocalHeapWriteBehindQueue<K, V>(
                    executionService, defaultThreadPool,
                    config, cacheLoaderWriter));
  }
}
```

可以看到，会创建 concurrency level 个队列 NonBatchingLocalHeapWriteBehindQueue，其又通过如下代码片段创建线程池和线程池队列。

```
this.executorQueue =
        new LinkedBlockingQueue<Runnable>(config. getMaxQueueSize());
if (config.getThreadPoolAlias() == null) {
  this.executor = executionService.getOrderedExecutor(
                                   defaultThreadPool, executorQueue);
} else {
  this.executor = executionService.getOrderedExecutor(
                          config. getThreadPoolAlias(), executorQueue);
}
```

- **CacheLoaderWriter**：此处我们只配置了write和delete，而writeAll和deleteAll将会把批量操作委托给write和delete。

PooledExecutionService#getOrderedExecutor 方法会创建 PartitionedOrderedExecutor 实例。

```
PartitionedOrderedExecutor(
        BlockingQueue<Runnable> queue, ExecutorService executor) {
  this.delegate = new PartitionedUnorderedExecutor(queue, executor, 1);
}
```

其使用 maxWorkers=1 创建了 PartitionedUnorderedExecutor，然后 PartitionedUnorderedExecutor 通过 this.runnerPermit = new Semaphore(maxWorkers)来控制并发，即

maxWorkers=1 就实现了一个并发。

因此，Ehcache 实际能写的最大队列大小为 concurrency level * queue size。

因为内部使用线程池去写，因此就实现了异步写，又因为使用了队列，因此控制了总的吞吐量（此处要注意根据实际场景给线程池配置 Rejected Policy），接下来看一下如何实现批量写。

2. 批量写

```
.withLoaderWriter(new DefaultCacheLoaderWriter<String, String>() {
    @Override
    public void writeAll(Iterable<? extends Map.Entry<? extends String, ? extends String>> entries) throws BulkCacheWritingException, Exception {
        for(Object entry : entries) {
            //batch write
        }
    }
    @Override
    public void deleteAll(Iterable<? extends String> keys) throws BulkCacheWritingException, Exception {
        for(Object key : keys) {
            //batch delete
        }
    }
})
.add(WriteBehindConfigurationBuilder
        .newBatchedWriteBehindConfiguration(3, TimeUnit.SECONDS, 2)
        .queueSize(5)
        .concurrencyLevel(1)
        .enableCoalescing()
        .useThreadPool("writeBehindPool")
        .build()));
```

和上一个示例不同的地方是使用了 newBatchedWriteBehindConfiguration 进行批量配置。

- newBatchedWriteBehindConfiguration(long maxDelay, TimeUnit maxDelayUnit, int batchSize)：设置批处理大小和最大延迟。batchSize用于定义批处理大小，当写操作数量等于批处理大小时，将把这一批数据发给CacheLoaderWriter进行处理。Ehcache使用BatchingLocalHeapWriteBehindQueue实现批量队列，其中操作批量的代码如下。

```
if (openBatch.add(operation)) {//往batch里添加操作，添加的数量等于批处理大小时
  submit(openBatch); //异步提交批处理操作
  openBatch = null;
}
```

因此，Ehcache 实际能写的最大队列大小为 concurrency level * queue size * batch size。

maxDelay 用于配置未完成的批处理最大延迟，比如，我们设置批处理大小为 3，而我们实际只写入了两个数据，当写第 3 个数据时，会触发提交批处理操作。但是，如果我们不写第 3 个，那么将造成这 2 个数据一直等待，我们可以设置 maxDelay，当超时时也会将这两个数据提交批处理。

- **enableCoalescing**：是否需要合并写，即对于相同的key只记录最后一次数据。
- **CacheLoaderWriter**：write和delete会转换为writeAll和deleteAll，即批处理。

9.6.6 Copy Pattern

有两种 Copy Pattern，Copy-On-Read（在读时复制）和 Copy-On-Write（在写时复制），在 Guava Cache 和 Ehcache 中堆缓存都是基于引用的，这样如果有人拿到缓存数据并修改了它，则可能发生不可预测的问题，笔者就见过因为这种情况造成数据错误。Guava Cache 没有提供支持，Ehcache 3.x 提供了支持。

```
public interface Copier<T> {
  T copyForRead(T obj); //Copy-On-Read，比如myCache.get()
  T copyForWrite(T obj); //Copy-On-Write，比如myCache.put()
}
```

通过如下方法来配置 key 和 Value 的 Copier。

```
CacheConfigurationBuilder.withKeyCopier()
CacheConfigurationBuilder.withValueCopier()
```

9.7 性能测试

笔者使用 JMH 1.14 进行基准性能测试，比如测试写。

```
@Benchmark
@Warmup(iterations = 10, time = 10, timeUnit = TimeUnit.SECONDS)
```

```
@Measurement(iterations = 10, time = 10, timeUnit = TimeUnit.SECONDS)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(1)
public void test_1_Write() {
    counterWriter = counterWriter + 1;
    myCache.put("key" + counterWriter, "value" + counterWriter);
}
```

使用 JMH 时首先进行 JVM 预热，然后进行度量，产生测试结果（本文使用吞吐量）。建议读者按照需求进行基准性能测试，以选择适合自己的缓存框架。

9.8 参考资料

[1] http://www.ehcache.org/documentation/3.1/caching-terms.html

[2] http://www.ehcache.org/documentation/3.1/caching-concepts.html

[3] http://www.ehcache.org/documentation/3.1/caching-patterns.html

[4] http://www.ehcache.org/documentation/3.1/clustered-cache.html

[5] http://www.ehcache.org/documentation/3.1/writers.html

[6] http://www.ehcache.org/documentation/3.1/serializers-copiers.html

[7] https://github.com/google/guava/wiki/CachesExplained

[8] http://mapdb.org/doc/

[9] http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/

10 HTTP缓存

10.1 简介

遇到很多人咨询笔者关于浏览器缓存的一些问题，而这些问题都是类似的，本章内容就是用来解答以后会遇到的类似问题。

因本章主要以浏览器缓存场景来介绍，所以非浏览器场景下的一些用法本章不会介绍，而且本章以 Chrome 为测试浏览器。

浏览器缓存是指当我们使用浏览器访问一些网站页面或者 HTTP 服务时，根据服务器端返回的缓存设置响应头将响应内容缓存到浏览器，下次可以直接使用缓存内容或者仅需要去服务器端验证内容是否过期即可。这样的好处是可以减少浏览器和服务器端之间来回传输的数据量，节省带宽以提升性能。

首先看个例子，当我们第一次访问 http://item.jd.com/1856588.html 时，将得到如下响应头。

```
▼ General
    Request URL: http://item.jd.com/1856588.html
    Request Method: GET
    Status Code: ● 200 OK
    Remote Address: 111.206.231.1:80
▼ Response Headers     view source
    Age: 3
    Cache-Control: max-age=60
    Connection: keep-alive
    Content-Encoding: gzip
    Content-Type: text/html; charset=gbk
    Date: Tue, 16 Aug 2016 12:07:44 GMT
    Expires: Tue, 16 Aug 2016 12:08:23 GMT
    Last-Modified: Tue, 16 Aug 2016 12:07:25 GMT
    ser: 3.81
    Server: jdws
    Transfer-Encoding: chunked
    Vary: Accept-Encoding
    Via: BJ-Y-NX-111(HIT), http/1.1 BJ-UNI-1-JCS-116 ( [cSsSfU])
```

接着按 F5 刷新页面，将得到如下响应头。

```
▼ General
    Request URL: http://item.jd.com/1856588.html
    Request Method: GET
    Status Code: ● 304 Not Modified
    Remote Address: 111.206.231.1:80
▼ Response Headers     view source
    Cache-Control: max-age=60
    Connection: keep-alive
    Date: Tue, 16 Aug 2016 12:07:53 GMT
    Expires: Tue, 16 Aug 2016 12:08:23 GMT
    Server: jdws
    Vary: Accept-Encoding
    Via: http/1.1 BJ-UNI-1-JCS-116 ( [cHs f ])
```

第二次返回的相应状态码为 304，表示服务器端文档没有修改过，浏览器缓存的内容还是最新的。

接下来，我们看一下如何在 Java 应用层控制浏览器缓存。

10.2 HTTP 缓存

10.2.1 Last-Modified

如下是我们的 spring mvc 缓存测试代码。

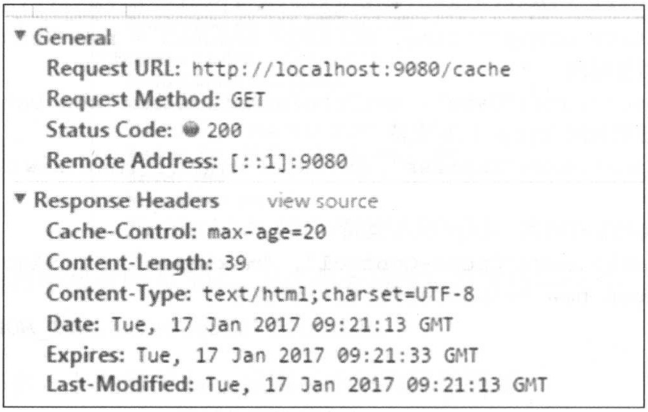
```
@RequestMapping("/cache")
public ResponseEntity<String> cache(
        //浏览器验证文档内容是否为修改时传入的 Last-Modified
        @RequestHeader(value = "If-Modified-Since", required = false)
        Date ifModifiedSince) throws Exception {

    DateFormat gmtDateFormat =
        new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss 'GMT'", Locale.US);

    //文档最后修改时间（去掉毫秒值）（为方便测试，每 10 秒生成一个新的）
    long lastModifiedMillis = getLastModified() / 1000 * 1000;

    //当前系统时间（去掉毫秒值）
    long now = System.currentTimeMillis() / 1000 * 1000;
    //文档可以在浏览器端/proxy 上缓存多久（单位：秒）
    long maxAge = 20;

    //判断内容是否修改了，此处使用等值判断
    if (ifModifiedSince != null
            && ifModifiedSince.getTime() == lastModifiedMillis) {
        MultiValueMap<String, String> headers = new HttpHeaders();
        //当前时间
        headers.add("Date", gmtDateFormat.format(new Date(now)));
        //过期时间 http 1.0 支持
        headers.add("Expires", gmtDateFormat.format(new Date(now + maxAge
* 1000)));
        //文档生存时间 http 1.1 支持
        headers.add("Cache-Control", "max-age=" + maxAge);
        return new ResponseEntity<String>(
                            headers, HttpStatus.NOT_MODIFIED);
    }

    String body = "<a href=''>点击访问当前链接</a>";
    MultiValueMap<String, String> headers = new HttpHeaders();
    //当前时间
    headers.add("Date", gmtDateFormat.format(new Date(now)));
    //文档修改时间
    headers.add("Last-Modified", gmtDateFormat.format(new
Date(lastModifiedMillis)));
    //过期时间 http 1.0 支持
    headers.add("Expires", gmtDateFormat.format(new Date(now + maxAge *
1000)));
    //文档生存时间 http 1.1 支持
    headers.add("Cache-Control", "max-age=" + maxAge);
```

```
        return new ResponseEntity<String>(body, headers, HttpStatus.OK);
    }

    Cache<String, Long> lastModifiedCache =
                CacheBuilder.newBuilder()
                    .expireAfterWrite(10, TimeUnit.SECONDS).build();

    public long getLastModified() throws ExecutionException {
        return lastModifiedCache.get("lastModified",
                        () -> { return System.currentTimeMillis(); });
    }
```

为了方便测试，文档的修改时间每十秒更新一次，实际应用时可以使用如商品的最后修改时间等替代。

1. 首次访问

首次访问 http://localhost:9080/cache，将得到如下响应头。



▼ General
Request URL: http://localhost:9080/cache
Request Method: GET
Status Code: ● 200
Remote Address: [::1]:9080
▼ Response Headers view source
Cache-Control: max-age=20
Content-Length: 39
Content-Type: text/html;charset=UTF-8
Date: Tue, 17 Jan 2017 09:21:13 GMT
Expires: Tue, 17 Jan 2017 09:21:33 GMT
Last-Modified: Tue, 17 Jan 2017 09:21:13 GMT

响应状态码 200 表示请求内容成功，另外，有如下几个缓存控制参数。

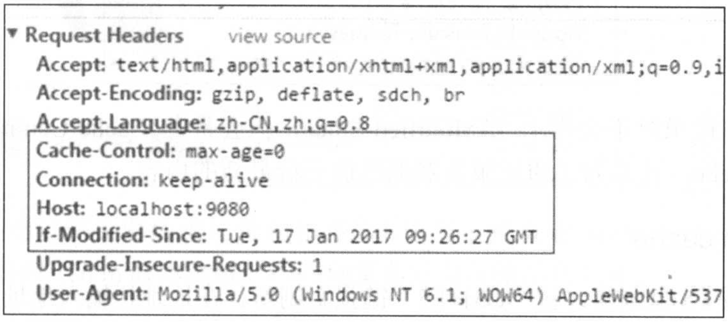
- **Last-Modified**：表示文档的最后修改时间，当去服务器验证时会用到这个时间。
- **Expires**：http/1.0规范定义，表示文档在浏览器中的过期时间，当缓存内容时间超过这个时间，则需要重新去服务器获取最新的内容。
- **Cache-Control**：http/1.1规范定义，表示浏览器缓存控制，max-age=20表示文档可以在浏览器中缓存20秒。

根据规范定义 Cache-Control 优先级高于 Expires。实际使用时可以两个都用，或仅使用 Cache-Control 就可以了（比如京东的活动页 sale.jd.com）。一般情况下 **Expires=当**

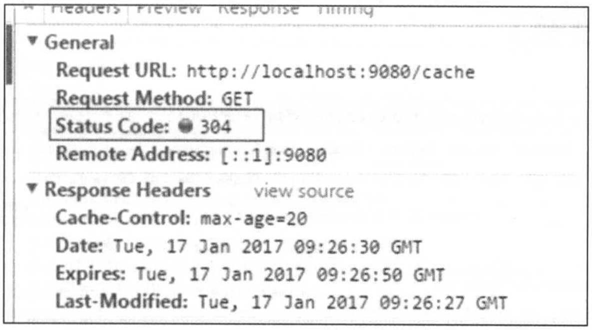
前系统时间（Date） + 缓存时间毫秒值（Cache-Control: max-age）。大家可以在如上测试代码中对两者进行单独测试，缓存都是可行的。

2. F5 刷新

接着按 F5 键刷新当前页面，将看到浏览器发送如下请求头。



此处发送时有一个 If-Modified-Since 请求头，其值是上次请求响应中的 Last-Modified，即浏览器会用这个时间去服务器端验证内容是否发生了变更。接着收到如下响应信息。



响应状态码为 304，表示服务器端通知浏览器“你缓存的内容没有变化，直接使用缓存内容展示吧”。

注：在测试时，要过一段时间就更改一下参数 millis，以表示内容修改了，要不然会一直看到 304 响应。

3. Ctrl+F5 强制刷新

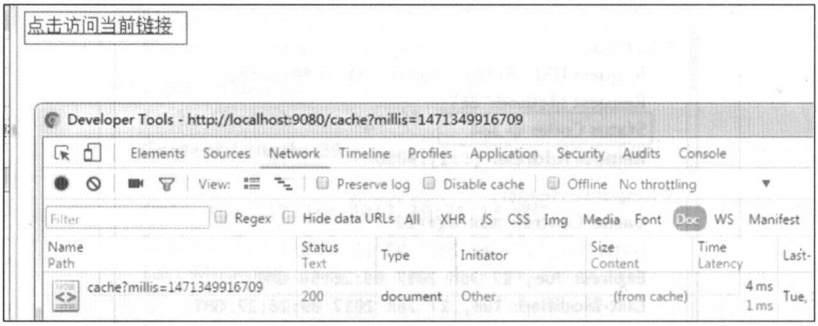
如果你想强制从服务器端获取最新的内容，则可以按“Ctrl+F5”组合键。

浏览器在请求时不会带上 If-Modified-Since，但会带上 Cache-Control:no-cache 和 Pragma:no-cache，这是为了通知服务器端提供一份最新的内容。

4. from cache

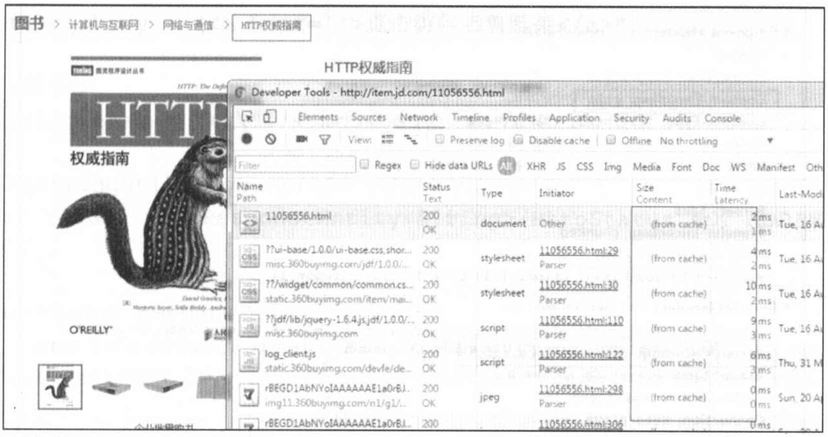
当我们按 F5 键刷新、按“Ctrl+F5”键强制刷新、在地址栏输入地址刷新时，都会去服务器端验证内容是否发生了变更。那什么情况才不去服务器端验证呢？有些朋友还会发现有一些“from cache”的问题，这是什么情况下发生的呢？

答案都是，从 A 页面跳转到 A 页面或者从 A 页面跳转到 B 页面时。



大家可以自行模拟从 A 页面跳转到 A 页面。此时，如果内容还在缓存时间之内，则直接从浏览器获取内容，而不去服务器端验证。

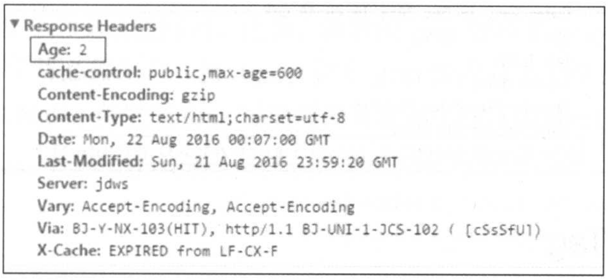
访问页面 http://item.jd.com/11056556.html，然后点击面包屑中的 HTTP 权威指南时，会跳转到当前页面，此时看到如下结果，页面及页面异步加载的一些 js、css、图片都 from cache 了。



还有，如通过浏览器历史记录进行前进后退时，也会走 from cache。本文是基于 Chrome 52.0.2743.116 m 版本测试，不同浏览器行为可能存在差异。

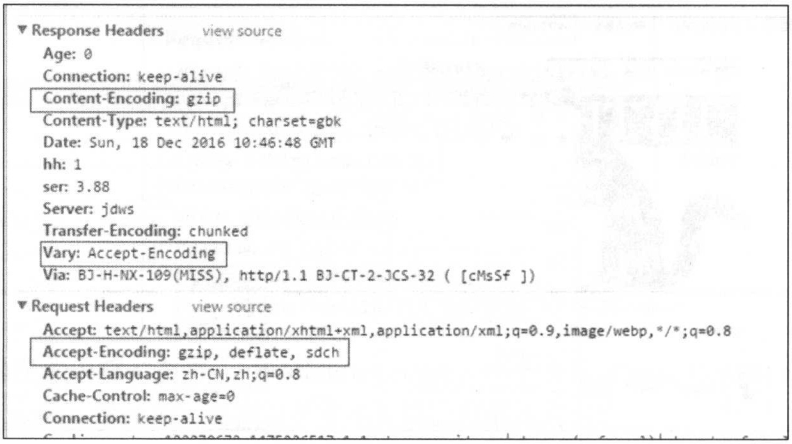
5. Age

一般用于缓存代理层（如 CDN）。大家在访问京东一些页面时，会发现有一个 Age 响应头，强制刷新（Ctrl+F5）后会发现其不断变化。这表示此内容在缓存代理层从创建到现在生存了多长时间。



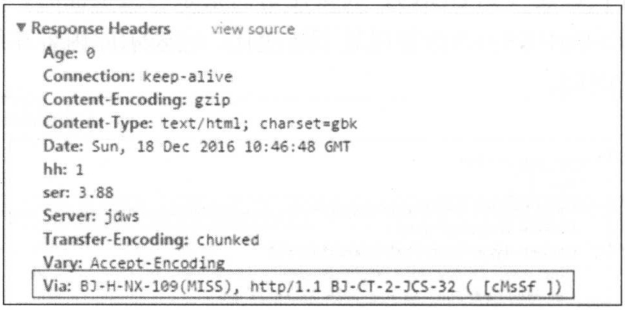
6. Vary

一般用于缓存代理层（如 CDN），如响应头列表，如“Vary:Accept-Encoding”、“Vary:User-Agent”，主要用于通知缓存服务器对于相同 URL 有着不同版本的响应，比如压缩版本和非压缩版本。缓存服务器应该根据 Vary 头来缓存不同版本的内容，如指定响应头为“Vary:Accept-Encoding”，则缓存代理层需要根据“Accept-Encoding”请求头来判断不同版本缓存内容，如“Accept-Encoding:gzip”请求头版本、无 Accept-Encoding 请求头版本。



7. Via

一般用于代理层（如 CDN），表示访问到最终内容前经过了哪些代理层，用的什么协议，代理层是否缓存命中等。通过它可以进行一些故障诊断。



10.2.2 ETag

```
@RequestMapping("/cache/etag")
public ResponseEntity<String> cache(
        //浏览器验证文档内容的实体 If-None-Match
        @RequestHeader (value = "If-None-Match", required = false)
        String ifNoneMatch) {

    //当前系统时间
    long now = System.currentTimeMillis();
    //文档可以在浏览器端/proxy上缓存多久
    long maxAge = 10;
```

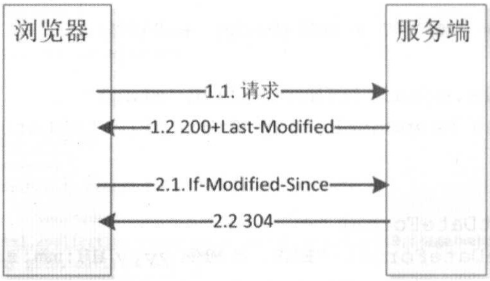
```
    String body = "<a href=''>点击访问当前链接</a>";

    //弱实体
    String etag = "W/\"" + md5(body) + "\"";

    if(StringUtils.equals(ifNoneMatch, etag)) {
        return new ResponseEntity<String>(HttpStatus.NOT_MODIFIED);
    }

   DateFormat gmtDateFormat =
       new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss 'GMT'", Locale.US);
    MultiValueMap<String, String> headers = new HttpHeaders();

    //ETag http 1.1 支持
    headers.add("ETag", etag);
    //当前系统时间
    headers.add("Date", gmtDateFormat.format(new Date(now)));
    //文档生存时间 http 1.1 支持
    headers.add("Cache-Control", "max-age=" + maxAge);
    return new ResponseEntity<String>(body, headers, HttpStatus.OK);
}
```

其中，ETag 是用于发送到服务器端进行内容变更验证的，而 Catch-Control 是用于控制缓存时间的（浏览器、代理层等）。此处我们使用了弱实体 W\"343sda"，弱实体（"343sda"）只要内容语义没变即可。比如，内容的 gzip 版和非 gzip 版可以使用弱实体验证；而强实体指字节必须完全一致（gzip 和非 gzip 情况是不一样的），因此，建议首先选择使用弱实体。Nginx 在生成 Etag 时使用的算法是 Last-Modified + Content-Length。

```
ngx_sprintf(etag->value.data,"\"%xT-%xO\"",
                                    r->headers_out.last_modified_time,
                                    r->headers_out.content_length_n)
```

到此简单的基于文档修改时间和过期时间的缓存控制就介绍完了。在内容型响应中，我们大多数根据内容的修改时间来进行缓存控制，ETag 根据实际需求而定（比如图片/JS/CSS 就非常适合 ETag，而如商品详情页使用商品修改时间，Last-Modified 处理更简单一些）。另外，还可以使用 html Meta 标签控制浏览器缓存，但是，对代理层缓存无效，因此不建议使用。

10.2.3 总结



1. 服务器端响应的 Last-Modified 会在下次请求时，将 If-Modified-Since 请求头带到服务器端进行文档是否修改的验证，如果没有修改则返回 304，浏览器可以直接使用缓存内容。

2. Cache-Control:max-age 和 Expires 用于决定浏览器端内容缓存多久，即多久过期，过期后则删除缓存重新从服务器端获取最新的。另外，可以用于 from cache 场景。

3. HTTP/1.1 规范定义的 Cache-Control 优先级高于 HTTP/1.0 规范定义的 Expires。

4. 一般情况下 Expires=当前系统时间 + 缓存时间（Cache-Control:max-age）。

5. HTTP/1.1 规范定义 ETag 为"被请求变量的实体值"，可简单理解为文档内容摘要，ETag 可用来判断页面内容是否已经被修改过了。

Last-Modified 与 ETag 同时使用时，浏览器在验证时会同时发送 If-Modified-Since 和 If-None-Match。按照 HTTP/1.1 规范，如果同时使用 If-Modified-Since 和 If-None-Match，则服务器端必须两个都验证通过后才能返回 304，Nginx 就是这样做的。因此，实际使用时应该根据实际情况选择。If-Match 和 If-Unmodified-Since 不作介绍。

10.3 HttpClient 客户端缓存

HttpClient 4.3 版本开始提供 HTTP/1.1 兼容的客户端缓存（HTTP/1.0 缓存没有实现），可以把该层看成浏览器缓存。HttpClient 通过职责链模式来支持可插拔的组件结构，客

户端缓存就是通过该模式实现的。有了此实现，直接开箱即用，不需要额外写代码来实现缓存。

在使用 HttpClient 客户端缓存时，需要引入如下依赖。

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient-cache</artifactId>
    <version>4.5.2</version>
</dependency>
```

本文使用 HttpClient 4.5.2 版本。在使用时，要通过如下配置创建 HttpClient。

```
CacheConfig cacheConfig = CacheConfig.custom()
        .setMaxCacheEntries(1000) //最多缓存 1000 个条目
        .setMaxObjectSize(1 * 1024 * 1024) //缓存对象最大为 1MB
        .setAsynchronousWorkersCore(5) //异步更新缓存线程池最小空闲线程数
        .setAsynchronousWorkersMax(10) //异步更新缓存线程池最大线程数
        .setRevalidationQueueSize(10000) //异步更新线程池队列大小
        .build();
//缓存存储
HttpCacheStorage cacheStorage = new BasicHttpCacheStorage(cacheConfig);
//创建 HttpClient
httpClient = CachingHttpClients.custom()
        .setCacheConfig(cacheConfig) //缓存配置
        .setHttpCacheStorage(cacheStorage) //缓存存储
        .setSchedulingStrategy(new          ImmediateSchedulingStrategy
(cacheConfig)) //验证缓存时，缓存调度策略
        .setConnectionManager(manager)
        .build();
```

如上配置省略了一些无关配置，CachingHttpClients 用于创建带客户端缓存的 HttpClient，其他配置请参考 HttpClient 连接池配置章节。

CacheConfig 主要进行如下几个方面的配置。

- **maxCacheEntries**：缓存条目数量，当缓存的数量超了会进行清除。
- **maxObjectSize**：每个缓存对象的最大大小，超过该大小的内容将不会被缓存，主要目的是防止出现缓存过大的内容。
- **asynchronousWorkersCore/asynchronousWorkersMax/revalidationQueueSize**：异步更新缓存内容线程池相关配置。

此外，HttpCacheStorage 用于指定 HTTP 响应内容使用什么存储器来存储，

BasicHttpCacheStorage 表示放在内存中存储（使用 LinkedHashMap 实现了最简单 LRU 算法）。默认还提供了 Ehcache 和 Memcached 存储实现。其 BasicHttpCacheStorage 没有基于时间的过期策略，建议实际使用时根据需要选择如 Ehcache 或者自己扩展一个实现（比如，扩展后支持多级缓存：堆内存→本地磁盘→分布式）。

SchedulingStrategy 用于配置当缓存需要重新验证时使用的异步调度策略，默认使用 ImmediateSchedulingStrategy，将使用我们配置的线程池参数创建线程池，然后异步进行重新验证请求。

接下来我们看看使用代码怎么实现。

```
HttpGet get = new HttpGet("http://sale.jd.com/act/qXdphQWLoFS.html?spm=1.1.0");
get.addHeader("User-Agent", "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.116 Safari/537.36");
HttpCacheContext context = HttpCacheContext.create();
CloseableHttpResponse response = getHttpClient().execute(get, context);
//缓存状态
CacheResponseStatus cacheResponseStatus = context. getCacheResponseStatus();
```

缓存状态有 HIT（响应命中，返回缓存的响应内容，不会发送请求到上游服务器）、MISS（缓存未命中，响应来自上游服务器）、VALIDATED（缓存不新鲜需要重新到上游服务器验证，且验证后返回缓存中的响应）、MODULE_RESPONSE（缓存直接生成的响应，比如，请求头“Cache-Control: only-if-cached”表示只使用缓存内容，但是如缓存没有，则生成一个 504 响应，此时缓存状态为 MODULE_RESPONSE）。

当我们多次调用如上代码后会发现，第一次访问时会是 MISS，第二次则会是 HIT。当然，前提是上游服务器设置了缓存响应头。

HttpClient 请求流程如下。

1．检查 HTTP 请求是否符合 HTTP/1.1 规范，如果不符合，则会进行修正（比如，请求头 Cache-Control 同时配置了 **max-age** 和 **no-cache**）。

2．清除该请求中的无效请求头。

3．检查该请求是否可以使用缓存内容，如果不能则发送请求到上游服务器获取新的内容。

4．如果该请求可以使用缓存内容作为响应，则尝试读取缓存中的缓存内容。如果读取失败，则同样发送请求到上游服务器获取最新的内容。

5. 如果缓存的响应内容可以使用，则会构建一个包含 ByteArrayEntity 的 BasicHttpResonse 对象。否则，会向上游服务器发出重新验证缓存内容的请求。

6. 如果缓存的响应内容向上游服务器验证失败，那么会重新向上游服务器发出一次不含缓存头的请求来获取最新的内容。

HttpClient 响应流程如下。

1. 检查收到的响应是否兼容 HTTP/1.1，如果不兼容，则会让其符合规范。

2. 检查响应是否可以缓存，如果可以，则会从响应中读取内容体，并缓存起来。

3. 如果响应数据太大，超出了配置的大小，则直接返回响应不进行缓存。

10.3.1 主流程

HttpClient 使用了职责链模式实现，使用 CachingExec 组件进行缓存相关操作，流程代码如下。

```
//使请求符合规范
requestCompliance.makeRequestCompliant(request);
// Via 头，请求每经过一层代理将会添加代理服务器标识，
//通过该头就知道经过了多少代理服务器
request.addHeader("Via",via);
//清除无效的缓存内容
flushEntriesInvalidatedByRequest(context.getTargetHost(), request);
//如果当前不能走缓存，则直接访问上游服务器
if (!cacheableRequestPolicy.isServableFromCache(request)) {
    log.debug("Request is not servable from cache");
    return callBackend(route, request, context, execAware);
}
//从缓存获取内容
final HttpCacheEntry entry = satisfyFromCache(target, request);
if (entry == null) {//如果没有命中，则调用 handleCacheMiss
    return handleCacheMiss(route, request, context, execAware);
} else {//如果命中了，则调用 handleCacheHit
    return handleCacheHit(route, request, context, execAware, entry);
}
```

10.3.2 清除无效缓存

接下来，让我们先看一下 flushEntriesInvalidatedByRequests 是怎么实现的，其会调

用 HttpCacheInvalidator# flushInvalidatedCacheEntries 进行清除当前请求相关的无效缓存：

```
//生成缓存 key
final String theUri = cacheKeyGenerator.getURI(host, req);
final HttpCacheEntry parent = getEntry(theUri);//获取缓存的内容
//如果请求方法不是“GET”或“HEAD”，或者当前请求是“GET”，但是，
//缓存中的请求方法是“HEAD”
//那么将需要清除无效缓存
if (requestShouldNotBeCached(req)
        || shouldInvalidateHeadCacheEntry(req, parent)) {
    if (parent != null) {
        //失效当前请求的不同版本（URL 相同，Vary 响应头不同）
        for (final String variantURI : parent.getVariantMap().values()) {
            flushEntry(variantURI);
        }
        //失效当前请求
        flushEntry(theUri);
    }
    final URL reqURL = getAbsoluteURL(theUri);
    //如果 Authority 部分一样，则失效“Content-Location”指定 URL 的缓存
    final Header clHdr = req.getFirstHeader("Content-Location");
    if (clHdr != null) {
        final String contentLocation = clHdr.getValue();
        if (!flushAbsoluteUriFromSameHost(reqURL, contentLocation)) {
            flushRelativeUriFromSameHost(reqURL, contentLocation);
        }
    }
    final Header lHdr = req.getFirstHeader("Location");
    if (lHdr != null) {//失效“Location”指定 URI 的缓存
        flushAbsoluteUriFromSameHost(reqURL, lHdr.getValue());
    }
}
```

此处使用 CacheKeyGenerator 生成缓存 key，key 格式为 protocol + hostname+ port + path + "?" + query。

10.3.3 查找缓存

接下来我们看一下 **CacheableRequestPolicy**#isServableFromCache 方法，其用于判断当前是否可以使用缓存。

```
public boolean isServableFromCache(final HttpRequest request) {
    final String method = request.getRequestLine().getMethod();
    final ProtocolVersion pv =
```

```
                request.getRequestLine(). getProtocolVersion();
    //如果请求不是 HTTP/1.1，则不能走缓存
    if (HttpVersion.HTTP_1_1.compareToVersion(pv) != 0) {
        return false;
    }
    //如果请求方法不是“GET”或者“HEAD”方法，不能走缓存
    if (!(method.equals(HeaderConstants.GET_METHOD) || method
            .equals(HeaderConstants.HEAD_METHOD))) {
        return false;
    }
    //如果请求头有“Pragma”，则不能走缓存
    if (request.getHeaders(HeaderConstants.PRAGMA).length > 0) {
        return false;
    }
    //如果请求头“Cache-Control”为“no-store”或者“no-cache”，则不能走缓存
    final Header[] cacheControlHeaders =
                  request.getHeaders (HeaderConstants.CACHE_CONTROL);
    for (final Header cacheControl : cacheControlHeaders) {
        for (final HeaderElement cacheControlElement :
                                       cacheControl.getElements()) {
            if ("no-store"
                    .equalsIgnoreCase(cacheControlElement.getName())) {
                return false;
            }
            if ("no-cache"
                    .equalsIgnoreCase(cacheControlElement.getName())) {
                return false;
            }
        }
    }
    return true;
}
```

此处只是请求不能使用缓存，并不会影响请求响应的缓存。另外，HttpClient 目前只对 HTTP/1.1 提供客户端缓存支持。

satisfyFromCache 方法委托 BasicHttpCache#getCacheEntry 方法从缓存中获取缓存内容。

```
  //首先根据 URL 缓存 key 获取缓存内容
  final HttpCacheEntry root = storage.getEntry(uriExtractor.getURI(host,
request));
  if (root == null) {//如果没有缓存，则直接返回
    return null;
```

```
    }
    if (!root.hasVariants()) {//如果缓存不存在，则是Vary版本的变体
        return root;
    }
    //获取Vary版本的URL缓存key
    final String variantCacheKey = root.getVariantMap().get (uriExtractor.
getVariantKey(request, root));
    if (variantCacheKey == null) {
        return null;
    }
    //获取并使用Vary版本的URL缓存
    return storage.getEntry(variantCacheKey);
```

之前部分已经解释了 Vary，此处也得到了验证。

10.3.4 缓存未命中

如果客户端缓存没有命中，则调用 handleCacheMiss 方法执行未命中逻辑。

```
    //如果有请求头“Cache-Control: only-if-cached”，则表示请求只从缓存中获取，
    //未命中情况返回504状态码
    if (!mayCallBackend(request)) {
        return Proxies.enhanceResponse(
                new BasicHttpResponse(
                        HttpVersion.HTTP_1_1,
                        HttpStatus.SC_GATEWAY_TIMEOUT,
                        "Gateway Timeout"));
    }
    //获取带Etag版本的Vary版本URL，如果有，则调用negotiateResponseFromVariants
    final Map<String, Variant> variants = getExistingCacheVariants(target,
request);
    if (variants != null && !variants.isEmpty()) {
        return negotiateResponseFromVariants(route, request, context,
                execAware, variants);
    }
    //否则调用callBackend回源到上游服务器
    return callBackend(route, request, context, execAware);
```

10.3.5 缓存命中

接下来，我们先看一下客户端缓存命中后调用 handleCacheHit 方法执行的命中逻辑。

```
    //用于判断缓存的响应是否可以直接返回给客户户
    if (suitabilityChecker.canCachedResponseBeUsed(target, request, entry,
```

```
now)) {
        //命中后生成缓存响应
        out = generateCachedResponse(request, context, entry, now);
        //缓存不新鲜需要验证，如果请求头有“Cache-Control: only-if-cached”,
        //则表示强制从缓存中获取，因此返回 504 响应
    } else if (!mayCallBackend(request)) {
        out = generateGatewayTimeout(context);
        //缓存不新鲜需要验证，如果缓存响应的状态码不是 304,
        //或者条件请求有请求头“"If-None-Match”或“If-Modified-Since”,
        //则需要请求上游服务器进行重新验证
    } else if (!(entry.getStatusCode() == HttpStatus.SC_NOT_MODIFIED
                    && !suitabilityChecker.isConditional(request))) {
        return revalidateCacheEntry(route, request, context, execAware, entry,
now);
    } else {
        //其他情况，直接回源上游服务器获取最新内容
        return callBackend(route, request, context, execAware);
    }
```

首先，看 CachedResponseSuitabilityChecker#canCachedResponseBeUsed 方法，其用于判断是否可以使用缓存的内容作为响应。

```
    //如果缓存的内容不是新鲜的，则不能使用缓存
    if (!isFreshEnough(entry, request, now)) {
        return false;
    }
    //如果请求方法是“GET”且缓存中的响应头“Content-Length”
    //与缓存 Body 体实际长度不一样，则可能内容不完整，不能使用缓存
    if (isGet(request)
            && !validityStrategy
                .contentLengthHeaderMatchesActualLength(entry)) {
        return false;
    }
    //如果请求头有 If-Range、If-Match、If-Unmodified-Since,
    //则 HttpClient 不支持，不能使用缓存
    if (hasUnsupportedConditionalHeaders(request)) {
        return false;
    }
    //如果缓存响应状态码为 304，但不是条件请求
    //（没有请求头 If-None-Match/ If-Modified-Since），则不能使用缓存
    if (!isConditional(request)
            && entry.getStatusCode() == HttpStatus. SC_NOT_MODIFIED) {
        return false;
    }
```

```
//如果是条件请求(请求时，业务代码自己设置了 If-None-Match/ If-Modified-Since，
//不建议这样，除非有具体理由)，但是条件请求与缓存中的响应头（Etag/Last-Modified）
//不匹配（如果两个都存在，则两个要都匹配才可以），则不能使用缓存
if (isConditional(request)
        && !allConditionalsMatch(request, entry, now)) {
    return false;
}
//如果缓存内容的请求方法或内容体为空，或者是一个 204 响应，则不能使用缓存
if (hasUnsupportedCacheEntryForGet(request, entry)) {
    return false;
}
for (final Header ccHdr :
                request.getHeaders(HeaderConstants. CACHE_CONTROL)) {
    for (final HeaderElement elt : ccHdr.getElements()) {
        //如果请求头有“Cache-Control:no-cache”，则不能使用缓存，需要回源验证
        if (HeaderConstants.CACHE_CONTROL_NO_CACHE
                            .equals(elt.getName())) {
            return false;
        }
        //如果请求头有“Cache-Control:no-store”，则不能使用缓存
        if (HeaderConstants.CACHE_CONTROL_NO_STORE
                            .equals(elt.getName())) {
            return false;
        }
        //如果请求头有“Cache-Control:max-age=time”，且当前 Age > max-age，
        //则不能使用缓存
        if (HeaderConstants.CACHE_CONTROL_MAX_AGE
                            .equals(elt.getName())) {
            final int maxage = Integer.parseInt(elt.getValue());
            if (validityStrategy.getCurrentAgeSecs(entry, now) > maxage) {
                return false;
            }
        }
        //如果请求头有“Cache-Control:max-stale=time”，
        //且保鲜时间 > max-stale（最大陈旧时间），这说明内容陈旧了
        //（最大陈旧时间要比保鲜时间更长才对），则不能使用缓存
        if (HeaderConstants.CACHE_CONTROL_MAX_STALE
                            .equals(elt.getName())) {
            final int maxstale = Integer.parseInt(elt.getValue());
            if (validityStrategy.getFreshnessLifetimeSecs(entry)
                                    > maxstale) {
                return false;
            }
        }
```

```
        //如果请求头有“Cache-Control:min-fresh=time”,
        //且（保鲜期-当前Age） < min-fresh（最小保鲜时间),
        //这说明内容不新鲜，则不能使用缓存
        if (HeaderConstants.CACHE_CONTROL_MIN_FRESH
                                        .equals(elt.getName())) {
            final long minfresh = Long.parseLong(elt.getValue());
            if (minfresh < 0L) {
                return false;
            }
            final long age =
                    validityStrategy.getCurrentAgeSecs(entry, now);
            final long freshness =
                     validityStrategy.getFreshnessLifetimeSecs(entry);
            if (freshness - age < minfresh) {
                return false;
            }
        }
    }
}
return true;
```

- **请求时间：** 表示HttpClient创建请求的时间。
- **响应时间：** 表示HttpClient接收到响应的时间。
- **响应头Date：** 表示上游服务器创建内容的时间，跟随响应返回给客户端。
- **接收时Age：** 如果有代理缓存服务器，则响应头会有Age，表示内容在缓存服务器已经存在了多久。还可以通过（响应时间-响应头Date）来计算初始时间，这两个时间取最大的一个。
- **当前Age：** 内容的当前生存期，即内容已经存在多久了，等于“接收时Age”+“响应延迟”（响应时间–请求时间）+“当前系统时间-响应时间”。
- **保鲜时间：** 即内容允许缓存的最大时间，为“Cache-Control:max-age”、“Cache-Control:s-maxage”或“Expires–Date”。

下面是 isFreshEnough 方法的实现。

```
//如果当前Age<保鲜时间，表示缓存响应是新鲜的
if (getCurrentAgeSecs(entry, now) < getFreshnessLifetimeSecs(entry)) {
   return true;
}
//如果使用启发式缓存（默认没开启），且判断是新鲜的，则返回缓存内容给客户
if (useHeuristicCaching &&
      validityStrategy.isResponseHeuristicallyFresh(
         entry, now, heuristicCoefficient, heuristicDefaultLifetime)) {
```

```
    return true;
}
//如果缓存响应有响应头“Cache-Control:must-revalidate”,则内容需要重新验证,
//即不新鲜
if (validityStrategy.mustRevalidate(entry)) {
    return false;
}
//如果开启共享缓存（只缓存 public 的，private 不缓存）
//如果缓存响应有响应头“Cache-Control: proxy-revalidate”或者“Cache-Control:
//s-maxage”，则认为内容不新鲜
if (sharedCache) {
    if(validityStrategy.proxyRevalidate(entry) ||
         validityStrategy.hasCacheControlDirective(entry, "s-maxage")) {
        return false;
}
}
//当前 Age-保鲜时间为内容已陈旧时间，即不新鲜多久了
//如果请求中的最大陈旧时间 > 已陈旧时间，则说明允许返回陈旧不新鲜的内容
final long maxstale = getMaxStale(request);
if (maxstale == -1) {//maxstale 等于-1 表示没有设置陈旧时间，认为内容陈旧不新鲜了
    return false;
}
long stalenessSecs = 0L;
final long age = getCurrentAgeSecs(entry, now);
final long freshness = getFreshnessLifetimeSecs(entry);
if (age <= freshness) {
    stalenessSecs  = 0L;
} else {
    stalenessSecs = (age - freshness);
}
return (maxstale > stalenessSecs);
```

10.3.6 缓存内容陈旧需重新验证

接来下，看一下 CachingExec#revalidateCacheEntry 实现。

```
//如果创建了异步验证器（当配置了 asynchronousWorkersMax>0 时会创建）
//如果允许陈旧的响应，且前往上游服务器验证时允许返回陈旧的响应，则使用异步后台验证
if (asynchRevalidator != null
      && !staleResponseNotAllowed(request, entry, now)
      && validityPolicy.mayReturnStaleWhileRevalidating(entry, now)) {
   final CloseableHttpResponse resp =
                 generateCachedResponse(request, context, entry, now);
   asynchRevalidator.revalidateCacheEntry(
```

```
                        this, route, request, context, execAware, entry);
    return resp;
}
//否则，同步验证
return revalidateCacheEntry(route, request, context, execAware, entry);
```

CachingExec#staleResponseNotAllowed 表示哪些情况必须前往上游服务器验证，不能返回陈旧的内容。

- 如果缓存的响应有响应头“Cache-Control:**must-revalidate**”。
- 如果开启了共享缓存，且缓存的响应有响应头“Cache-Control: **proxy-revalidate**”。
- 或者，如果缓存的响应有响应头“Cache-Control:**max-stale**”，当内容陈旧时——(当前Age-保鲜期)>最大陈旧时间。
- 如果有“Cache-Control:min-fresh”或者“Cache-Control:max-age”时（执行到此处说明内容已经不新鲜了，不满足新鲜条件）。

CacheValidityPolicy#mayReturnStaleWhileRevalidating 实现。

如果缓存响应的响应头有“**Cache-Control**: **stale-while-revalidate=time**”，此 time 指定在验证期间允许返回陈旧的响应（即可以异步后台验证），且已陈旧时间<=time，则可以异步验证，并返回陈旧的内容。

接下来，看一下 CachingExec#revalidateCacheEntry 如何进行验证（省略了一些非关键代码）。

```
//构建条件请求
final HttpRequestWrapper conditionalRequest = conditionalRequestBuilder.
        buildConditionalRequest(request, cacheEntry);
……
//请求时间
Date requestDate = getCurrentDate();
//执行条件请求
CloseableHttpResponse backendResponse =
      backend.execute(
             route, conditionalRequest, context, execAware);
//响应时间
Date responseDate = getCurrentDate();
……
//如果响应太旧（响应时间早于缓存的响应头 Date 时间），则重新发出一次非条件查询重新获取
if (revalidationResponseIsTooOld(backendResponse, cacheEntry)) {
   backendResponse.close();
```

```
        final HttpRequestWrapper unconditional = conditionalRequestBuilder
            .buildUnconditionalRequest(request, cacheEntry);
        requestDate = getCurrentDate();
        backendResponse =
                backend.execute(route, unconditional, context, execAware);
        responseDate = getCurrentDate();
    }

    final int statusCode = backendResponse.getStatusLine().getStatusCode();
    if (statusCode == HttpStatus.SC_NOT_MODIFIED
                || statusCode == HttpStatus.SC_OK) {
        //缓存状态为 VALIDATED
        setResponseStatus(context, CacheResponseStatus.VALIDATED);
    }

    if (statusCode == HttpStatus.SC_NOT_MODIFIED) {
        //如果响应返回 304，则说明内容没有变化
        final HttpCacheEntry updatedEntry = responseCache.updateCacheEntry(
                context.getTargetHost(), request, cacheEntry,
                backendResponse, requestDate, responseDate);
        //如果当前请求是条件查询，且“Etag/Last-Modified”都匹配，则生成 304 响应
        if (suitabilityChecker.isConditional(request)
                && suitabilityChecker.allConditionalsMatch(
                                    request, updatedEntry, new Date())) {
            return responseGenerator
                    .generateNotModifiedResponse(updatedEntry);
        }
        //正常响应
        return responseGenerator.generateResponse(request, updatedEntry);
    }
    ……
    //处理后端响应，比如缓存响应
    return handleBackendResponse(conditionalRequest, context, requestDate,
                                responseDate, backendResponse);
```

ConditionalRequestBuilder#buildConditionalRequest 用于构建条件请求。

```
    //复制请求
    final HttpRequestWrapper newRequest =
                    HttpRequestWrapper.wrap(request. getOriginal());
    //复制请求头
    newRequest.setHeaders(request.getAllHeaders());
    //如果缓存响应中有 Etag，则将 Etag 设置到请求头 If-None-Match
    final Header eTag = cacheEntry.getFirstHeader(HeaderConstants.ETAG);
    if (eTag != null) {
```

```
    newRequest.setHeader(
                    HeaderConstants.IF_NONE_MATCH, eTag.getValue());
}
//如果缓存响应中有 Last-Modified,
//则将 Last-Modified 设置到请求头 If-Modified-Since
final Header lastModified =
             cacheEntry.getFirstHeader (HeaderConstants.LAST_MODIFIED);
if (lastModified != null) {
    newRequest.setHeader(
          HeaderConstants.IF_MODIFIED_SINCE, lastModified.getValue());
}
boolean mustRevalidate = false;
for(final Header h :
                cacheEntry.getHeaders(HeaderConstants.CACHE_ CONTROL)) {
    for(final HeaderElement elt : h.getElements()) {
        if (HeaderConstants.CACHE_CONTROL_MUST_REVALIDATE
                                  . equalsIgnoreCase(elt.getName())
          || HeaderConstants.CACHE_CONTROL_PROXY_REVALIDATE
                                  . equalsIgnoreCase(elt.getName())) {
          mustRevalidate = true;
          break;
        }
    }
}
//如果必须强制验证(类似于浏览器按了 F5 键),则设置请求头“Cache-Control:max-age=0”
if (mustRevalidate) {
    newRequest.addHeader(
                HeaderConstants.CACHE_CONTROL,
                HeaderConstants.CACHE_CONTROL_MAX_AGE + "=0");
}
return newRequest;
```

10.3.7 缓存内容无效需重新执行请求

接下来，看一下 CachingExec#callBackend 实现。

```
//执行请求
final CloseableHttpResponse backendResponse =
                backend.execute(route, request, context, execAware);
//添加 Via 头
backendResponse.addHeader("Via", generateViaHeader(backendResponse));
//处理后端响应，如缓存响应
return handleBackendResponse(request, context, requestDate,
                                  getCurrentDate(), backendResponse);
```

10.3.8 缓存响应

最后我们来看一下 CachingExec#handleBackendResponse 逻辑。

```
//判断响应是否可以缓存
final boolean cacheable = responseCachingPolicy.isResponseCacheable (
                                        request, backendResponse);
//清除过期的缓存条目
responseCache.flushInvalidatedCacheEntriesFor(
                                target, request, backendResponse);
//如果可以缓存，并且缓存中没有更新的缓冲条目（比如被别的线程更新了），则缓存响应
if (cacheable &&
      !alreadyHaveNewerCacheEntry(target, request, backendResponse)) {
    //如果响应状态码为 304，如果有请求头 If-Modified-Since，
    //则将其添加到响应头 Last-Modified
   storeRequestIfModifiedSinceFor304Response(request, backendResponse);
   return responseCache.cacheAndReturnResponse(target, request,
                       backendResponse, requestDate, responseDate);
}
//如果不缓存，则清除缓存条目
if (!cacheable) {
    responseCache.flushCacheEntriesFor(target, request);
}
……
```

ResponseCachingPolicy#isResponseCacheable 用于判断什么情况下可以缓存。

- 如果请求协议版本不是HTTP/1.1，则不能缓存。
- 如果“Cache-Control:no-store”则不能缓存（no-cache是可以缓存的，但是每次需要验证）。
- 如果 URL 中有“？”，且 CacheConfig 配置了 neverCacheHTTP10ResponsesWithQuery=true（表示从不缓存带参数的HTTP/1.0响应），如果响应头Via有“HTTP/1.0”或“1.0”，则表示中间代理层有HTTP/1.0的版本，不能缓存。
- 如果URL中有“?”，且没有缓存头“Expires”或“Cache-Control”为max-age、s-maxage、must-revalidate、proxy-revalidate、public，则不能缓存。
- 如果缓存头“Expires”早于请求头“Date”，表示内容已陈旧，则不能缓存。
- 如果开启了共享缓存（sharedCache），如果有请求头“Authorization”，但是响应头“Cache-Control”没有s-maxage、must-revalidate或public时，则不能缓存。
- 如果请求方法不是GET、HEAD，则不能缓存。
- 如果响应状态码不为200、203、300、301、410，则不能缓存。

- 如果Content-Length>maxObjectSize，则不能缓存。
- 如果有多个Age、Expires头，则不能缓存。
- 如果Date头不为1个，或者Date无法解析，则不能缓存。
- 如果Vary是*，则不能缓存。
- 如果Cache-Control为no-store、no-cache、或者private，但开启了共享缓存，则不缓存。

在默认情况下，HttpClient CacheConfig 参数（isSharedCache=true）表示只缓存公有缓存(public)，不会缓存带有授权头“Authorization”（除非明确指定为 public）或 private 缓存，可以设置 isSharedCache=false 来关闭共享缓存。

Heuristic Cache，即启发式缓存，即使服务器没有明确设置缓存控制头时，它也会缓存一定数目的响应，默认是关闭的。如果需要，则可以配置 CacheConfig 的 heuristicCachingEnabled、heuristicCoefficient、heuristicDefaultLifetime 相关参数开启。

到此我们介绍完了 HttpClient 缓存的主要内容，其中忽略了一些不影响主流程的细节，如果想要彻底理解，则建议深入阅读源码。

10.3.9 缓存头总结

根据如上的源码分析，我们再总结一下 Cache-Control。

- **public**：响应头，可共享缓存（客户端和代理服务器都可以缓存），响应可以被缓存。
- **private**：响应头，可私有缓存（客户端可以缓存，代理服务器不能缓存），比如用户私有内容，不能共享。
- **no-cache**：请求头使用时表示需要回源验证，响应头使用时表示允许缓存者缓存响应，但是，使用时必须回源验证，所以此处叫no-cache并不是很好。
- **no-store**：请求和响应禁止缓存。
- **max-age**：缓存的保鲜期和Expires类似，根据该值校验缓存是否新鲜。
- **s-maxage**：与max-age的区别是其仅用于共享缓存（如缓存代理服务器），当客户端缓存不新鲜时遇到此头要重新验证。
- **max-stale**：缓存的最大陈旧时间，如果缓存不新鲜但还在该最大陈旧时间内，则可以返回陈旧的内容。
- **min-fresh**：缓存的最小新鲜期，请求时使用（保鲜期–当前Age） < min-fresh 判断内容是否新鲜。

- **must-revalidate**：当缓存过了新鲜期后，必须回源重新验证。与no-cache类似，但是更严格，不能使用后台重新验证，而no-cache允许后台重新验证。
- **proxy-revalidate**：与must-revalidate类似，但是，只对缓存代理服务器有效，客户端遇到此头需要回源重新验证。
- **stale-while-revalidate**：请求时，表示在指定的时间内可以先返回陈旧的内容，后台进行重新验证（如异步验证）。
- **stale-if-error**：请求时，表示在指定的时间内，当重新验证请求响应状态码为500、502、503、504时，可以使用陈旧内容。
- **only-if-cached**：请求时，使用该头表示只从缓存获取响应，如果没有，则504 Gateway Timeout。

10.4 Nginx HTTP 缓存设置

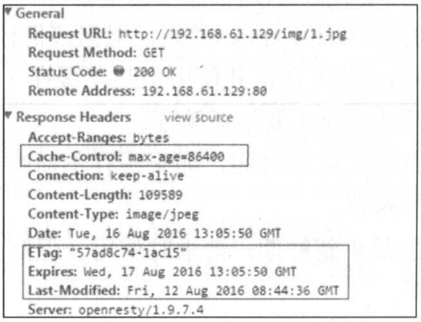
Nginx 提供了 expires、etag、if-modified-since 指令来实现浏览器缓存控制。

10.4.1 expires

如果我们使用 Nginx 作为静态资源服务器，那么可以使用 expires 进行缓存控制。

```
location /img {
  alias /export/img/;
  expires 1d;
}
```

当我们访问静态资源，如 http://192.168.61.129/img/1.jpg 时，将得到类似如下的响应头。



对于静态资源会自动添加 ETag，可以通过配置 etag off 指令禁止生成 ETag。如果是静态文件，那么 Last-Modified 值为文件的最后修改时间。Expires 是根据当前服务器系统时间算出来的。如下是 Nginx expires 配置的计算逻辑（实际计算逻辑要更多，请参考官方文档）。

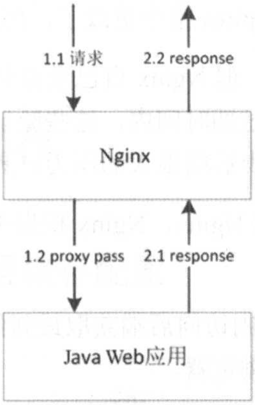
```
if (expires == NGX_HTTP_EXPIRES_ACCESS ||r->headers_out.last_modified_
time == -1) {
    max_age = expires_time;
    expires_time += now;
}
```

10.4.2 if-modified-since

此指令用于指定 Nginx 如何对服务器端的 Last-Modified 和浏览器端的 if-modified-since 时间进行比较，默认的“if_modified_since exact”表示精确匹配，也可以使用“if_modified_since _before”表示只要文件的最后修改时间早于或等于浏览器端的 if-modified-since 时间，就返回 304。

10.4.3 nginx proxy_pass

使用 Nginx 作为反向代理时，请求会先进入 Nginx，然后 Nginx 将请求转发给后端应用，如下图所示。



首先配置 upstream。

```
upstream backend_tomcat {
    server 192.168.61.1:9080 max_fails=10 fail_timeout=10s weight=5;
}
```

接着配置 location

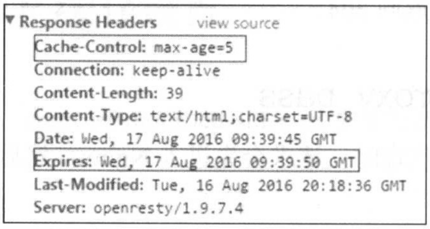
```
location = /cache {
   proxy_pass http://backend_tomcat/cache$is_args$args;
}
```

接下来，我们可以通过如 http://192.168.61.129/cache?millis=1471349916709 访问 Nginx，Nginx 会将请求转发给后端 Java 应用。也就是说 Nginx 只是做了相关的转发（负载均衡），并没有对请求和响应做什么处理。

假设需要对后端返回的过期时间进行调整，可以添加 Expires 指令到 location。

```
location = /cache {
   proxy_pass http://backend_tomcat/cache$is_args$args;
   expires 5s;
}
```

然后再请求相关的 URL，将得到如下响应。



过期时间相关的响应头被 Expires 指令更改了，但是 last-modified 是没有变的。

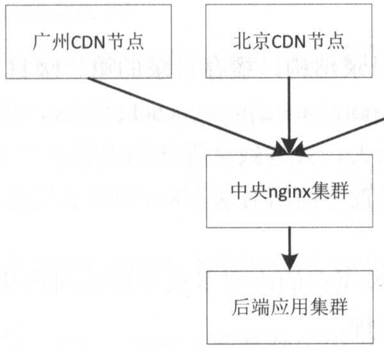
即使我们更改了缓存过期头，但 Nginx 自己没有对这些内容做代理层缓存，每次请求还是要到后端验证的。假设在过期时间内，这些验证在 Nginx 这一层进行就可以了，不需要到后端验证，这样可以减少后端很大的压力。具体整体流程如下。

1. 浏览器发起请求，首先到 Nginx，Nginx 根据 URL 在 Nginx 本地查找是否有代理层本地缓存。

2. Nginx 没有找到本地缓存，则访问后端获取最新的文档，并放入 Nginx 本地缓存，返回 200 状态码和最新的文档给浏览器。

3. Nginx 找到本地缓存，首先验证文档是否过期（Cache-Control:max-age=5）。如果过期，则访问后端获取最新的文档，并放入 Nginx 本地缓存，返回 200 状态码和最新的文档给浏览器；如果文档没有过期，即 if-modified-since 与缓存文档的 last-modified 匹配，则返回 304 状态码给浏览器。

内容不需要访问后端，即不需要后端动态计算/渲染等，直接 Nginx 代理层就把内容返回，速度更快——内容越接近于用户速度越快。像 Apache Traffic Server、Squid、Varnish、Nginx 等技术都可以用来进行内容缓存。还有 CDN 技术就是用来加速用户访问的。



用户首先访问全国各地的 CDN 节点（使用如 ATS、Squid 实现），如果 CDN 没命中，则会回源到中央 Nginx 集群，该集群做二级缓存，如果没有命中缓存（该集群的缓存不是必须的，要根据实际命中情况等决定），则最后回源到后端应用集群。

像我们商品详情页的一些服务就大量使用了 Nginx 缓存减少回源到后端的请求量，从而提升访问速度。可以参考“第 11 章 多级缓存”、“第 16 章 构建需求响应式亿级商品详情页”和“第 17 章 京东商品详情页服务闭环实践”。

10.5 Nginx 代理层缓存

10.5.1 Nginx 代理层缓存配置

1. HTTP 模块配置

```
proxy_buffering                on;
proxy_buffer_size              4k;
proxy_buffers                  512 4k;
proxy_busy_buffers_size        64k;
proxy_cache_path               /export/cache/proxy_cache    levels=1:2
keys_zone=cache:512m inactive=5m max_size=8g use_temp_path=off;
```

```
#proxy timeout
proxy_connect_timeout          3s;
proxy_read_timeout             5s;
proxy_send_timeout             5s;
```

proxy_cache_path 指令配置：

- **levels=1:2**：表示创建两级目录结构，缓存目录的第一级目录是1个字符，第二级目录是2个字符，比如/export/cache/proxy_cache/7/3c/，如果将所有文件放在一级目录下的话，文件量很大，会导致文件访问变慢。
- **keys_zone=cache:512m**：设置存储所有缓存key和相关信息的共享内存区，1M大约能存储8000个key。
- **inactive=5m**：inactive指定被缓存的内容多久不被访问将从缓存中移除，以保证内容的新鲜，默认为10分钟。
- **max_size=8g**：最大缓存阈值，“cache manager”进程会监控最大缓存大小，当缓存达到该阈值时，该进程将从缓存中移除最近最少访问的内容。
- **use_temp_path**：如果为on，则内容首先被写入临时文件（proxy_temp_path），然后重命名到proxy_cache_path指定的目录；如果设置为off，则内容直接被写入到proxy_cache_path指定的目录，如果需要cache建议off。（该特性是1.7.10提供的。）

2. proxy_cache 配置

```
location = /cache {
    proxy_cache cache;
    proxy_cache_key $scheme$proxy_host$request_uri;
    proxy_cache_valid 200 5s;
    proxy_pass http://backend_tomcat/cache$is_args$args;
   add_header cache-status $upstream_cache_status;
}
```

缓存相关配置。

- **proxy_cache**：指定使用哪个共享内存区存储缓存信息。
- **proxy_cache_key**：设置缓存使用的key，默认为完整的访问URL，根据实际情况设置缓存key。
- **proxy_cache_valid**：为不同的响应状态码设置缓存时间。如果是proxy_cache_valid 5s，则200、301、302响应都将被缓存。

proxy_cache_valid 不是唯一设置缓存时间的，还可以通过如下方式（优先级从

上到下）实现。

- 以秒为单位的“X-Accel-Expires”响应头来设置响应缓存时间。
- 如果没有“X-Accel-Expires”，则可以根据“Cache-Control”、“Expires”来设置响应缓存时间。
- 否则，使用 proxy_cache_valid 设置缓存时间。

如果响应头包含 Cache-Control：private/no-cache/no-store、Set-Cookie，或者只有一个 Vary 响应头且其值为*，则响应内容将不会被缓存。可以使用 proxy_ignore_headers 来忽略这些响应头。

- **add_header cache-status $upstream_cache_status**在响应头中添加缓存命中的状态。
  - **HIT：**缓存命中，直接返回缓存中内容，不回源到后端。
  - **MISS：**缓存未命中，回源到后端获取最新的内容。
  - **EXPIRED：**缓存命中但过期了，回源到后端获取最新的内容。
  - **UPDATING：**缓存已过期但正在被别的 Nginx Worker 进程更新，配置了 proxy_cache_use_stale updating 指令时会存在该状态。
  - **STALE：**缓存已过期，但因后端服务出现了问题（比如后端服务挂了）返回过期的响应，配置了如 proxy_cache_use_stale error timeout 指令后会出现该状态。
  - **REVALIDATED：**启用 proxy_cache_revalidate 指令后，当缓存内容过期时，Nginx 通过一次 if-modified-since 的请求头去验证缓存内容是否过期，此时会返回该状态。
  - **BYPASS：**proxy_cache_bypass 指令有效时，强制回源到后端获取内容，即使已经缓存了。
- **proxy_cache_min_uses：**用于控制请求多少次后响应才被缓存。默认的“proxy_cache_min_uses 1;”指，如果缓存热点比较集中、存储有限，则可以通过修改该参数来减少缓存数量和写磁盘次数。
- **proxy_no_cache：**用于控制什么情况下响应不被缓存。比如配置“proxy_no_cache $args_nocache”，如果带的nocache参数值至少有一个不为空或者为0，则响应将不被缓存。
- **proxy_cache_bypass：**类似于proxy_no_cache，但其控制什么情况不使用缓存的内容，而是直接到后端获取最新的内容。如果命中，则$upstream_cache_status为BYPASS。

- **proxy_cache_use_stale**：当对缓存内容的过期时间不敏感，或者后端服务出问题时，即使缓存的内容不新鲜也总比返回错误给用户强（类似于托底），此时可以配置该参数，如“proxy_cache_use_stale error timeout http_500 http_502 http_503 http_504”，即如果出现超时、后端连接出错、500/502/503等错误时，则即使缓存内容已过期也先返回给用户，此时$upstream_cache_status为STALE。还有一个updating表示缓存已过期但正在被别的Nginx Worker进程更新，只是先返回了过期内容，此时 $upstream_cache_status为UPDATING。
- **proxy_cache_revalidate**：当缓存过期后，如果开启了proxy_cache_revalidate，则会发出一次if-modified-since或if-none-match条件请求，如果后端返回304，则此时$upstream_cache_status为REVALIDATED，我们将得到两个好处，节省带宽和减少写磁盘的次数。
- **proxy_cache_lock**：当多个客户端同时请求同一份内容时，如果开启proxy_cache_lock（默认off），则只有一个请求被发送至后端。其他请求将等待该请求的返回。当第一个请求返回后，其他相同请求将从缓存中获取内容返回。当第一个请求超过了proxy_cache_lock_timeout超时时间（默认为5s），则其他请求将同时请求到后端来获取响应，且响应不会被缓存（在1.7.8版本之前是被缓存的）。启用proxy_cache_lock可以应对Dog-pile effect（当某个缓存失效时，同时有大量相同的请求没命中缓存，而同时请求到后端，从而导致后端压力太大，此时限制一个请求去获取即可）。
- **proxy_cache_lock_age**是1.7.8新添加的，如果在proxy_cache_lock_age指定的时间内（默认为5s），最后一个发送到后端进行新缓存构建的请求还没有完成，则下一个请求将被发送到后端来构建缓存（因为1.7.8版本之后，proxy_cache_lock_timeout超时之后返回的内容是不缓存的，需要下一次请求来构建响应缓存）。

10.5.2 清理缓存

有时缓存的内容是错误的，需要手工清理。Nginx 商业版提供了 purger 功能，对于社区版 Nginx，则可以考虑使用 ngx_cache_purge（https://github.com/FRiCKLE/ngx_cache_purge）模块进行缓存清理。

```
location ~ /purge(/.*) {
   allow               127.0.0.1;
   deny                all;
   proxy_cache_purge   cache$1$is_args$args;
```

```
}
```

该方法要限制访问权限，如只允许内网可以访问或者需要密码才能访问。

到此代理层缓存就介绍完了，通过代理层缓存可以解决很多问题，可以参考“第 17 章 京东商品详情页服务闭环实践”。

10.6 一些经验

- 只缓存200状态码的响应，像302等，要根据实际场景决定。比如，当系统出错时，自动302到错误页面，此时缓存302就不对了。
- 有些页面不需要强一致，可以进行几秒的缓存。比如商品详情页展示的库存，可以缓存几秒钟。短时间的不一致对于用户来说是没有影响的。
- JS/CSS/image等一些内容缓存时间可以设置为很久，比如1个月甚至1年，通过在页面修改版本来控制过期。
- 假设商品详情页异步加载的一些数据，使用last-modified进行过期控制，而服务器端做了逻辑修改，但内容是没有修改的，即内容的最后修改时间没变。如果想过期这些异步加载的数据，则可以考虑在商品详情页添加异步加载数据的版本号，通过添加版本号来加载最新的数据，或者将last-modified时间加1来解决，但这种情况下使用ETag是更好的选择。
- 商品详情页异步加载的一些数据，可以考虑更长时间的缓存，比如1个月而不是几分钟。可以通过MQ将修改时间推送到商品详情页，从而实现按需过期数据。
- 服务器端考虑使用tmpfs内存文件系统缓存、SSD缓存，使用服务器端负载均衡算法一致性哈希来提升缓存命中率。
- 缓存key要合理设计。比如，去掉某些参数或排序参数，以保证代理层的缓存命中率；要有清理缓存的工具，出问题时能快速清理掉问题key。
- AB测试/个性化需求时，要禁用掉浏览器缓存，但要考虑服务器端缓存。
- 为了便于查找问题，一般会在响应头中添加源服务器信息，如访问京东商品详情页会看到ser响应头，此头存储了源服务器IP，以便出现问题时，知道哪台服务器有问题。

参考资料

[1] https://www.w3.org/Protocols/HTTP/1.0/spec.html

[2] https://www.w3.org/Protocols/rfc2616/rfc2616.html

[3] http://nginx.org/en/docs/http/ngx_http_headers_module.html#expires

[4] http://nginx.org/en/docs/http/ngx_http_core_module.html#if_modified_since

[5] http://nginx.org/en/docs/http/ngx_http_core_module.html#etag

[6] http://nginx.org/en/docs/http/ngx_http_proxy_module.html

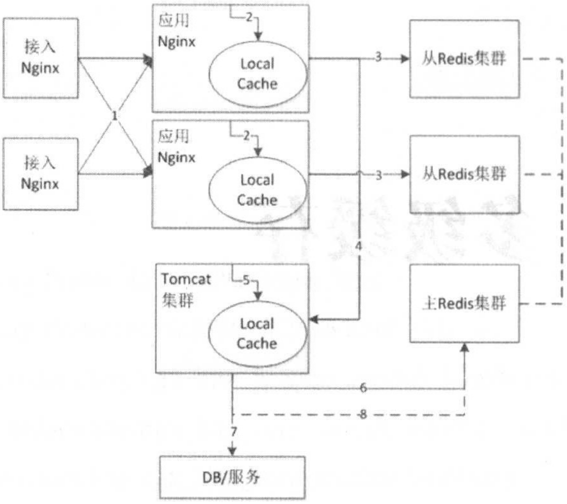
[7] http://hc.apache.org/httpcomponents-client-4.5.x/tutorial/html/index.html

11 多级缓存

缓存技术是一个老生常谈的话题，但是，它也是解决性能问题的利器，一把瑞士军刀。而且在各种面试过程中，或多或少会被问及一些缓存相关的问题，如缓存算法、热点数据与更新缓存、更新缓存与原子性、缓存崩溃与快速恢复等各种问题。而这些问题中，有些问题又是与场景相关，因此，如何合理应用缓存来解决问题也是一个选择题。本文所有内容都跟读服务缓存相关，不会涉及写服务数据的缓存。本文不考虑内容型应用前置的 CDN 架构，也不会涉及缓存数据结构优化、缓存空间利用率跟业务数据相关的细节问题，主要从架构和提升命中率等层面来探讨缓存方案。本文将基于多级缓存模式来介绍应用缓存时需要注意的问题和一些解决方案，其中一些方案已经实现，而有一些是正在尝试用来解决痛点问题。

11.1 多级缓存介绍

所谓多级缓存，是指在整个系统架构的不同系统层级进行数据缓存，以提升访问效率，这也是应用最广的方案之一。我们应用的整体架构和流程如下图所示。



整体流程如下。

1. 接入 Nginx 将请求负载均衡到应用 Nginx，此处常用的负载均衡算法是轮询或者一致性哈希。轮询可以使服务器的请求更加均衡，而一致性哈希可以提升应用 Nginx 的缓存命中率，后续在负载均衡和缓存算法部分我们再详细介绍。

2. 应用 Nginx 读取本地缓存（本地缓存可以使用 Lua Shared Dict、Nginx Proxy Cache（磁盘/内存）、Local Redis 实现）。如果本地缓存命中，则直接返回，使用应用 Nginx 本地缓存可以提升整体的吞吐量，降低后端压力，尤其应对热点问题非常有效。为什么要使用 Nginx 本地缓存我们将在热点数据与缓存失效部分详细介绍。

3. 如果 Nginx 本地缓存没命中，则会读取相应的分布式缓存（如 Redis 缓存，还可以考虑使用主从架构来提升性能和吞吐量），如果分布式缓存命中，则直接返回相应数据（并回写到 Nginx 本地缓存）。

4. 如果分布式缓存也没有命中，则会回源到 Tomcat 集群，在回源到 Tomcat 集群时，也可以使用轮询和一致性哈希作为负载均衡算法。

5. 在 Tomcat 应用中，首先读取本地堆缓存。如果有，则直接返回（并会写到主 Redis 集群），为什么要加一层本地堆缓存将在缓存崩溃与快速修复部分详细介绍。

6. 作为可选部分，如果步骤 4 没有命中，则可以再尝试一次读主 Redis 集群操作，目的是防止当从集群有问题时的流量冲击。

7. 如果所有缓存都没有命中，则只能查询 DB 或相关服务获取相关数据并返回。

8. 步骤 7 返回的数据异步写到主 Redis 集群，此处可能有多个 Tomcat 实例同时写

主 Redis 集群，会造成数据错乱，如何解决该问题将在更新缓存与原子性部分详细介绍。

整体分了三部分缓存：应用 Nginx 本地缓存、分布式缓存、Tomcat 堆缓存。每一层缓存都用来解决相关问题，如应用 Nginx 本地缓存用来解决热点缓存问题，分布式缓存用来减少访问回源率，Tomcat 堆缓存用于防止相关缓存失效/崩溃之后的冲击。

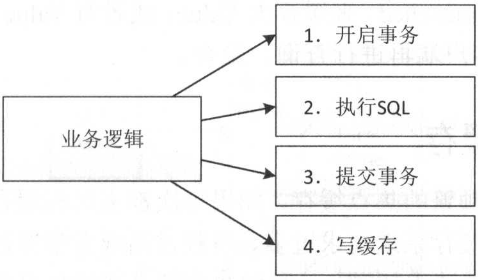
虽然都是加缓存，但是怎么加、怎么用，细想下来还是有很多问题需要权衡和考量的，接下来的部分我们就详细来讨论一些缓存相关的问题。

11.2 如何缓存数据

11.2.1 过期与不过期

对于缓存的数据我们可以考虑不过期缓存和带过期时间缓存，什么场景应该选择哪种模式需要根据业务和数据量等因素来决定。

不过期缓存场景一般思路如下图所示。



使用 Cache-Aside 模式，首先写数据库，如果成功，则写缓存。这种场景下存在事务成功、缓存写失败但无法回滚事务的情况。另外，不要把写缓存放在事务中，尤其写分布式缓存，因为网络抖动可能导致写缓存响应时间很慢，引起数据库事务阻塞。如果对缓存数据一致性要求不是那么高，数据量也不是很大，则可以考虑定期全量同步缓存。

为更好解决以上多个事务的问题，可以考虑使用“第 15 章 队列术”中所使用的基于 Canal 实现缓存同步。

对于长尾访问的数据、大多数数据访问频率都很高的场景，或者是缓存空间足够，都可以考虑不过期缓存，比如用户、分类、商品、价格、订单等。当缓存满了，可以考虑用 LRU 机制驱逐老的缓存数据。

过期缓存机制，如采用懒加载，一般用于缓存其他系统的数据（无法订阅变更消息，或者成本很高）、缓存空间有限、低频热点缓存等场景。常见步骤是首先读取缓存，如果不命中，则查询数据，然后异步写入缓存并设置过期时间，下次读取将命中缓存。热点数据经常使用过期缓存，即在应用系统上缓存比较短的时间。这种缓存可能存在一段时间的数据不一致情况，需要根据场景来决定如何设置过期时间。如库存数据可以在前端应用上缓存几秒钟，短时间的不一致是可以忍受的。

11.2.2 维度化缓存与增量缓存

对于电商系统，一个商品可能拆成如基础属性、图片列表、上下架、规格参数、商品介绍等。如果商品变更了，要把这些数据都更新一遍，更新成本很高，包括接口调用量和带宽。因此，最好将数据进行维度化并增量更新（只更新变的部分）。尤其如上下架这种只是一个状态变更但每天频繁调用的数据，维度化后能减少服务很大压力。

11.2.3 大 Value 缓存

要警惕缓存中的大 Value，尤其是使用 Redis 时。遇到这种情况时可以考虑使用多线程实现的缓存，如 Memcached，来缓存大 Value；或者对 Value 进行压缩；或者将 Value 拆分为多个小 Value，客户端再进行查询、聚合。

11.2.4 热点缓存

对于那些访问非常频繁的热点缓存，如果每次都去远程缓存系统中获取，可能会因为访问量太大导致远程缓存系统请求过多、负载过高或者带宽过高等问题，最终可能导致缓存响应慢，使客户端请求超时。一种解决方案是通过挂更多的从缓存，客户端通过负载均衡机制读取从缓存系统数据。不过也可以在客户端所在的应用/代理层本地存储一份，从而避免访问远程缓存，即使像库存这种数据，在有些应用系统中也可以进行几秒钟的本地缓存，从而降低远程系统的压力。

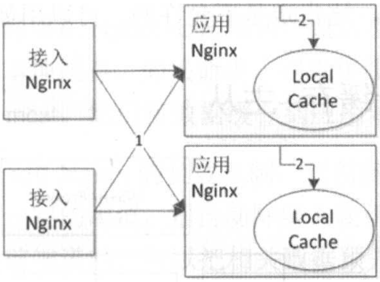
11.3 分布式缓存与应用负载均衡

11.3.1 缓存分布式

此处说的分布式缓存一般采用分片实现，即将数据分散到多个实例或多台服务器。算法一般采用取模和一致性哈希。要采用如之前所说的不过期缓存机制，可以考虑取模机制，扩容时一般是新建一个集群。而对于可以丢失的缓存数据，可以考虑一致性哈希，即使其中一个实例出问题只是丢一小部分，对于分片实现可以考虑客户端实现，或者使用如 Twemproxy 中间件进行代理（分片对客户端是透明的）。如果使用 Redis，则可以考虑使用 redis-cluster 分布式集群方案。

11.3.2 应用负载均衡

应用负载均衡一般采用轮询和一致性哈希，一致性哈希可以根据应用请求的 URL 或者 URL 参数将相同的请求转发到同一个节点。而轮询是将请求均匀地转发到每个服务器，如下图所示。



整体流程如下。

1．首先，请求进入接入层 Nginx。

2．根据负载均衡算法将请求转发给应用 Nginx。

3．如果应用 Nginx 本地缓存命中，则直接返回数据，否则读取分布式缓存或者回源到 Tomcat。

轮询的优点是，到应用 Nginx 的请求更加均匀，使得每个服务器的负载基本均衡。轮询的缺点是，随着应用 Nginx 服务器的增加，缓存的命中率会下降，比如，原来 10 台服务器命中率为 90%，再加 10 台服务器将可能降低到 45%。而这种方式不会因为热点问题导致其中某一台服务器负载过重。

一致性哈希的优点是，相同请求都会转发到同一台服务器，命中率不会因为增加服务器而降低。一致性哈希的缺点是，因为相同的请求会转发到同一台服务器，因此，可能造成某台服务器负载过重，甚至因为请求太多导致服务出现问题。

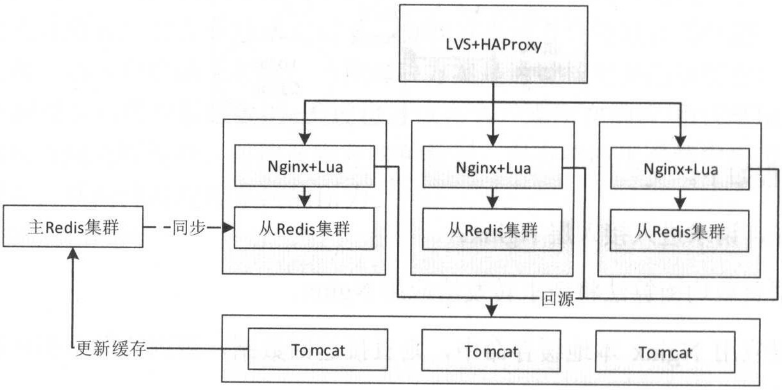
解决办法是根据实际情况动态选择使用哪种算法。

- 负载较低时，使用一致性哈希。
- 热点请求降级一致性哈希为轮询，或者如果请求数据有规律，则可考虑带权重的一致性哈希。
- 将热点数据推送到接入层Nginx，直接响应给用户。

11.4 热点数据与更新缓存

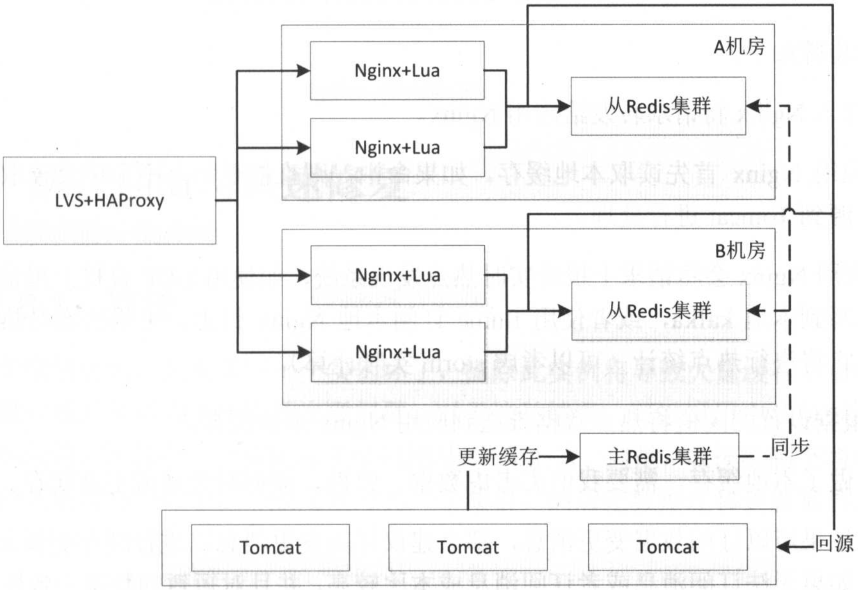
热点数据会造成服务器压力过大，导致服务器性能、吞吐量、带宽达到极限，出现响应慢或者拒绝服务的情况，这肯定是不允许的。可以用如下几个方案去解决。

11.4.1 单机全量缓存+主从



如上图所示，所有缓存都存储在应用本机，回源之后会把数据更新到主 Redis 集群，然后通过主从模式复制到其他从 Redis 集群。缓存的更新可以采用懒加载或者订阅消息进行同步。

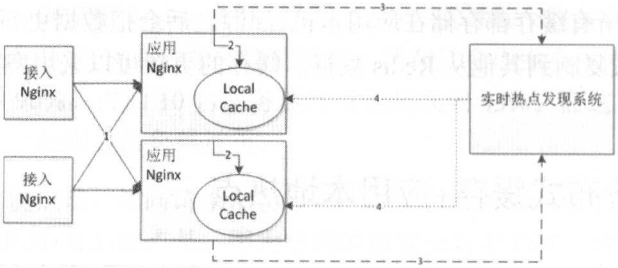
11.4.2 分布式缓存+应用本地热点



对于分布式缓存，我们需要在 Nginx+Lua 应用中进行应用缓存来减少 Redis 集群的访问冲击，即首先查询应用本地缓存，如果命中，则直接缓存，如果没有命中，则接着查询 Redis 集群、回源到 Tomcat，然后将数据缓存到应用本地。

对于 LVS+HAProxy 到应用 Nginx 的负载机制，正常情况采用一致性哈希，如果某个请求类型的访问量突破了一定的阈值，则自动降级为轮询机制。而对于一些秒杀活动之类的热点，我们是可以提前知道的，可以把相关数据预先推送到应用 Nginx，并将负载均衡机制降级为轮询。实际场景中我们是通过两级 Nginx（接入 Nginx→应用 Nginx）实现该特性的，没有在 LVS+HAProxy 层实现。

另外，可以考虑建立实时热点发现系统来发现热点。



具体步骤如下。

1．接入 Nginx 将请求转发给应用 Nginx。

2．应用 Nginx 首先读取本地缓存。如果命中，则直接返回，不命中会读取分布式缓存、回源到 Tomcat 进行处理。

3．应用 Nginx 会将请求上报给实时热点发现系统，如使用 UDP 直接上报请求，或者将请求写到本地 kafka，或者使用 flume 订阅本地 Nginx 日志。上报给实时热点发现系统后，它将进行热点统计（可以考虑 storm 实时计算）。

4．根据设置的阈值将热点数据推送到应用 Nginx 本地缓存。

因为做了本地缓存，需要我们去考虑数据一致性，即何时失效或更新缓存。

- 如果可以订阅数据变更消息，那么建议订阅变更消息以进行缓存更新。
- 如果无法订阅消息或者订阅消息成本比较高，并且对短暂的数据一致性要求不严格（比如，在商品详情页看到的库存，可以短暂的不一致，只要保证下单时一致即可），那么可以设置合理的过期时间，过期后再查询新的数据。
- 如果是秒杀之类的，可以订阅活动开启消息，将相关数据提前推送到前端应用，并将负载均衡机制降级为轮询。
- 建立实时热点发现系统来对热点进行统一推送和更新。

11.5 更新缓存与原子性

正如之前说的，如果多个应用同时操作一份数据，很可能导致缓存数据变成脏数据，解决办法如下。

- 更新数据时使用更新时间戳或者版本对比，如果使用Redis，则可以利用其单线程机制进行原子化更新。
- 使用如canal订阅数据库binlog。
- 将更新请求按照相应的规则分散到多个队列，然后每个队列进行单线程更新，更新时拉取最新的数据保存。
- 用分布式锁，在更新之前获取相关的锁。

11.6 缓存崩溃与快速修复

11.6.1 取模

对于取模机制，如果其中一个实例坏了，摘除此实例将导致大量缓存不命中，则瞬间大流量可能导致后端 DB/服务出现问题。对于这种情况，可以采用主从机制来避免实例坏了的问题，即其中一个实例坏了可以用从/主顶上来。但是，取模机制下增加一个节点将导致大量缓存不命中，一般是建立另一个集群，然后把数据迁移到新集群，把流量迁移过去。

11.6.2 一致性哈希

对于一致性哈希机制，如果其中一个实例坏了，摘除此实例只影响一致性哈希环上的部分缓存不命中，不会导致大量缓存瞬间回源到后端 DB/服务，但是也会产生一些影响。

另外，也可能因为一些误操作导致整个缓存集群出现问题，如何快速恢复呢？

11.6.3 快速恢复

如果出现之前说到的一些问题，可以考虑如下方案。

- 主从机制，做好冗余，即其中一部分不可用，将对等的部分补上去。
- 如果因为缓存导致应用可用性已经下降，可以考虑部分用户降级，然后慢慢减少降级量，后台通过Worker预热缓存数据。

也就是说，如果整个缓存集群坏了，而且没有备份，那么只能慢慢将缓存重建。为了让部分用户还是可用的，可以根据系统承受能力，通过降级方案让一部分用户先用起来，将这些用户相关的缓存重建。另外，通过后台 Worker 进行缓存数据的预热。

12 连接池线程池详解

在应用系统开发过程中，我们经常会用到池化技术，如对象池、连接池、线程池等，通过池化来减少一些消耗，以提升性能。对象池通过复用对象从而减少创建对象、垃圾回收的开销，但是，池不能太大，太大会影响 GC 时的扫描时间。连接池如数据库连接池、Redis 连接池、HTTP 连接池，通过复用 TCP 连接来减少创建和释放连接的时间来提升性能。线程池也是类似的，通过复用线程提升性能。也就是说池化的目的就是通过复用技术提升性能。

池化可以使用 Apache commons-pool 2 来实现，比如 DBCP、Jedis 连接池都是使用 commons-pool 2 实现的，最新的版本是 2.4.2。另外，不建议再使用 commons-pool 1.x 版本。而笔者也在工作中写过自己的连接池 fast-pool 以适应我们的场景。本文主要讲解数据库连接池 DBCP、HTTP 连接池 HttpClient 和线程池。

12.1 数据库连接池

数据库连接池有很多实现，如 C3P0、DBCP、Druid 等。笔者用得最多的是 Druid 和 DBCP。本文将以 commons-dbcp 2 2.1.1 作为示例进行讲解。

12.1.1 DBCP 连接池配置

```
<bean id="dataSource"
        class="org.apache.commons.dbcp2.BasicDataSource"
        destroy-method="close">
    <!-- 数据库连接相关配置（URL、用户名、密码、测试 Query、默认超时时间） -->
    <property name="url" value=""/>
    <property name="username" value=""/>
    <property name="password" value=""/>
    <!-- Statement 默认超时时间，单位：秒 -->
    <property name="defaultQueryTimeout" value="3"/>
    <!-- 默认是否自动提交事务，默认为 true -->
    <property name="defaultAutoCommit" value="false"/>

    <!—数据库连接属性（不同的数据库配置不一样）-->
    <property name="connectionProperties"
            value="connectTimeout=2000; socketTimeout=2000 "/>

    <!-- 连接池队列类型默认为 LIFO, false 表示 FIFO -->
    <property name="lifo" value="false"/>
    <!--建议以下值尽量一样，没必要频繁地过期空闲连接（除非出现连接池资源紧缺等情况，
才可以考虑） -->
    <property name="initialSize" value="80"/>
    <property name="minIdle" value="80"/>
    <property name="maxIdle" value="80"/>
    <property name="maxTotal" value="80"/>

    <!-- 这是等待获取连接池连接时间，也不要太大，比如设置在 500 毫秒 -->
    <property name="maxWaitMillis" value="500" />

    <!—验证数据库连接是否有效/可用 -->
    <!-- 从池中获取连接时进行 validateConnection，默认为 true -->
    <property name="testOnBorrow" value="true"/>
    <!-- 新建连接时进行 validateConnection，默认为 false -->
    <property name="testOnCreate" value="false"/>
    <!-- 将连接释放回池时进行 validateConnection，默认为 false -->
    <property name="testOnReturn" value="false"/>
    <!-- 如果不设置，则将调用 Connection#isValid(int timeout)验证数据库是否有
效 -->
    <property name="validationQuery" value=""/>
    <!-- 连接存活的最长时间，<=0 禁用该配置 -->
    <property name="maxConnLifetimeMillis" value="0"/>

    <!-- 驱除定时器执行周期，<=0 表示禁用 -->
```

```
    <property name="timeBetweenEvictionRunsMillis" value="0" />
    <!-- 连接空闲多久从池中驱除，<=0 不做判断 -->
    <!-- minIdle < 当前空闲连接数量，使用这个时间测试 -->
    <property name="softMinEvictableIdleTimeMillis" value="0"/>
    <!-- 连接空闲多久从池中驱除，与 softMinEvictableIdleTimeMillis 是或关系 -->
    <property name="minEvictableIdleTimeMillis" value="0" />
    <!-- 每次测试多少空闲对象，<=0 就相当于禁用 -->
    <property name="numTestsPerEvictionRun" value="0" />
    <!-- 当连接空闲时是否测试，即保持连接一直存活，配合驱除定时器使用 -->
    <property name="testWhileIdle" value="false"/>
    <!-- 判断连接是否需要驱除的策略，默认为 DefaultEvictionPolicy -->
    <property name="evictionPolicyClassName" value="org.apache.commons.
pool2.impl.DefaultEvictionPolicy"/>

    <!-- 移除无引用连接（那些没有 close 的连接），此处设置为 false，需要保证程序中连
接一定释放 -->
    <property name="removeAbandonedOnBorrow" value="false"/>
    <property name="removeAbandonedOnMaintenance" value="false"/>
    <!-- 超时后将自动关闭无引用连接，单位：秒 -->
    <property name="removeAbandonedTimeout" value="10"/>

</bean>
```

1. 数据库连接配置

配置数据库连接 URL（url）、用户名（username）、密码（password）、Statement 默认超时时间（defaultQueryTimeout）、事务自动提交（defaultAutoCommit）、数据库连接属性（connectionProperties，配置如连接超时时间等数据库特有属性，也可以在 JDBC URL 后面通过“?propName=propValue”配置 connectionProperties，更多参数请参考 http://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-configuration-properties.html）。

2. 池配置

配置连接池队列类型（lifo）是采用 LIFO 还是 FIFO 获取连接，默认为 FIFO。

其配置项包括初始大小（initialSize）、最小空闲大小（minIdle）、最大空闲大小（maxIdle）、最大大小（maxTotal）。连接如果不使用则会进入空闲，因此，空闲连接可以根据实际情况保持存活。如交易系统基本上 7×24 小时不停工作，可以考虑将如上的几个配置设置为一样，减少过期操作。

还有一个 maxWaitMillis，用于配置当数据库连接池没可用连接时的最大等待时间，

当超时后将抛出异常。

3. 验证数据库连接有效性

有三种办法：主动测试（创建连接时、获取连接时、释放连接时）、定时测试（通过定时器定期测试）、关闭孤儿连接。使用配置的 validationQuery（如果不配置默认调用 Connection#isValid 进行验证）和 maxConnLifetimeMillis（连接生存的最长时间）来验证连接是否可用。

主动测试

testOnBorrow 是获取连接时测试，testOnCreate 是创建连接时测试，testOnReturn 是释放连接时测试，测试代码如下。

```
public boolean validateObject(PooledObject<PoolableConnection> p) {
    try {
        validateLifetime(p);
        validateConnection(p.getObject());
        return true;
    } catch (Exception e) {
        return false; //表明当前连接要释放/销毁
    }
}

//验证连接的最大生存时间，如果配置了而且超出了最大生存期，则抛出异常
private void validateLifetime(PooledObject<PoolableConnection> p)
                                              throws Exception {
    if (maxConnLifetimeMillis > 0) {
        long lifetime = System.currentTimeMillis() - p.getCreateTime();
        if (lifetime > maxConnLifetimeMillis) {
            throw new LifetimeExceededException(Utils.getMessage(
                    "connectionFactory.lifetimeExceeded",
                    Long.valueOf(lifetime),
                    Long.valueOf(maxConnLifetimeMillis)));
        }
    }
}
//发送验证命令给底层数据库验证存活,
//validationQuery 要配置为至少返回一行记录的 SELECT 语句;
//如果不配置，则默认使用 Connection#isValid(int timeout)测试
public void validateConnection(PoolableConnection conn)
                                             throws SQLException {
    if(conn.isClosed()) {
```

```
        throw new SQLException("validateConnection: connection closed");
    }
    conn.validate(_validationQuery, _validationQueryTimeout);
}
```

MySQL Connector 也提供了 autoReconnect 和 autoReconnectForPools 配合 maxReconnects 来实现重连。

```
if ((this.autoReconnect.getValue())
            && (this.autoCommit || this. autoReconnectForPools.getValue())
            && this.needsPing && !isBatch) {
    try {
        pingInternal(false, 0);
        this.needsPing = false;
    } catch (Exception Ex) {
        createNewIO(true);
    }
}
```

在每次执行 SQL 之前根据配置进行一次 ping 测试。很多人在数据库连接 URL 上都配置了此参数，但 MySQL 官方不推荐这种做法，而是推荐通过如 DBCP2 使用 testOnBorrow 在获取连接时只进行一次测试。

定时测试

使用 timeBetweenEvictionRunsMillis 配置定时器执行周期，如果配置为<=0，则表示禁用，配置代码如下所示（BaseGenericObjectPool#startEvictor）。

```
if (delay > 0) {
    evictor = new Evictor();
    EvictionTimer.schedule(evictor, delay, delay);
}
```

EvictionTimer 是 static，因此一个 ClassLoader 将只有一个 Timer 进行调度，主要做以下两件事情。

```
try {
    evict();//1. 执行 Evict 任务
} catch(Exception e) {
    ……
}
try {
    ensureMinIdle();//2. 确保连接池最小空闲连接数
} catch (Exception e) {
```

```
        swallowException(e);
    }
```

Evict 任务主要有以下几件事情执行（GenericObjectPool#evict）。

```
//获取每次任务处理的连接数量（防止连接池配置过大，任务执行过长）
private int getNumTests() {
    int numTestsPerEvictionRun = getNumTestsPerEvictionRun();
    if (numTestsPerEvictionRun >= 0) {
        return Math.min(numTestsPerEvictionRun, idleObjects.size());
    } else {
        return (int) (Math.ceil(idleObjects.size() /
                Math.abs((double) numTestsPerEvictionRun)));
    }
}
```

接着会调用 EvictionPolicy（默认为 DefaultEvictionPolicy）判断当前连接是否需要被释放。

```
evict = evictionPolicy.evict(evictionConfig, underTest,
        idleObjects.size());
if (evict) {
    destroy(underTest);
}

DefaultEvictionPolicy#evict
public boolean evict(
        EvictionConfig config, PooledObject<T> underTest, int idleCount) {
    //如果当前连接的空闲时间大于 softMinEvictableIdleTimeMillis
    //且当前空闲连接大于配置的最小空闲连接，
    //或者当前连接的空闲时间大于 minEvictableIdleTimeMillis，则表示需要释放连接
    if ((config.getIdleSoftEvictTime() < underTest.getIdleTimeMillis()
                && config.getMinIdle() < idleCount)
            || config.getIdleEvictTime() < underTest.getIdleTimeMillis()) {
        return true;
    }
    return false;
}
```

可以配置 evictionPolicyClassName 来定义个性化释放策略。

如果配置了 testWhileIdle=true，则会调用 factory.validateObject(underTest)进行连接可用测试。

4. 关闭孤儿连接

如果获取连接后一直没有释放回池中，即该连接泄露了，如果不关闭的话，则会造成数据库连接被用完，因此，可以考虑配置 removeAbandoned*进行关闭孤儿连接。不过不建议配置，把代码写健壮吧。

12.1.2 DBCP 配置建议

如果并发量大则建议：几个池大小设置为一样，禁用关闭孤儿连接，禁用定时器。配置简化为如下代码。

```
<bean id="dataSource"
      class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close">
   <!—省略 url username password -->
   <property name="defaultQueryTimeout" value="3"/>
   <property name="defaultAutoCommit" value="false"/>
   <property name="connectionProperties"
             value="connectTimeout=2000; socketTimeout=2000 "/>
   <property name="testOnBorrow" value="true"/>
   <property name="lifo" value="false"/>
   <!—省略池配置（池大小设置为都一样） -->
   <property name="maxWaitMillis" value="500" />
   <property name="timeBetweenEvictionRunsMillis" value="0" />
</bean>
```

如果并发量不大则建议：可以按需设置池大小，禁用关闭孤儿连接，启用定时器（注意 MySQL 空闲连接 8 小时自动断开）。配置简化为如下代码。

```
<bean id="dataSource"
      class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close">
   <!—省略 url username password -->
   <property name="defaultQueryTimeout" value="3"/>
   <property name="defaultAutoCommit" value="false"/>
   <property name="connectionProperties"
             value="connectTimeout=2000; socketTimeout=2000 "/>
   <property name="testOnBorrow" value="false"/>
   <property name="lifo" value="false"/>
   <!—省略池配置（池大小设置为都一样） -->
   <property name="maxWaitMillis" value="500" />
   <property name="timeBetweenEvictionRunsMillis" value="3600000"/>
   <property name="numTestsPerEvictionRun" value="80" />
```

```
    <property name="testWhileIdle" value="true"/>
</bean>
```

不管你用哪种方式都要记得设置超时时间，在 JVM 关闭/重启时一定要销毁连接池（bean 配置 destroy-method="close"），因为如果没有加 destroy-method ，而且重启次数太频繁，将造成重启 tomcat 后旧的数据库连接池的连接不释放，这样会有很多数据库连接在一段时间内不释放，造成启动后无法建立连接。

如下是我们实际工程的配置，已经将参数都默认化了。

```
<ds:datasource
        id="orderDataSource"
        url="${mysql.order.url}"
        username="${mysql.order.username}"
        password="${mysql.order.password}"
        max-pool-size="50"/>
```

对于 MySQL，因为设置了 Statement 超时时间，超时则要杀掉 Statement。MySQL 通过 Timer 去完成这件事情，每个连接创建时会创建一个 Timer，执行 Statement 时给 Timer 分配一个任务，超时则要杀掉该 Statement。

```
timeoutTask = new StatementImpl.CancelTask(this);
//Timer 是惰性创建
conn.getCancelTimer().schedule(timeoutTask, (long)this.queryTimeout *
1000);
```

CancelTask 会启动一个新线程来执行如下逻辑。

```
if(queryTimeoutKillsConnection == true) {
  //force close connection
} else {
  //send "KILL QUERY ConnectionId" to mysql
}
```

如果我们配置了<property name="connectionProperties"value="queryTimeoutKillsConnection=true"/>，那么将强制关闭连接。默认 false，会通过创建一个新连接，然后执行“KILL QUERY connectionId”来杀掉此连接当前执行的 SQL。

12.1.3 数据库驱动超时实现

MySQL 驱动在创建每个连接时会创建一个 Timer（每个 Timer 是一个 Thread）。然后每个连接中创建的每个 Statement 会提交一个 TimerTask（超时则每个 Task 在执行时

会创建并启动一个新的 Thread)。

也就是说,假设一个数据库连接池创建了 500 个连接,每个连接执行 1 个 statement,最坏的情况下会创建:

500×1+500×1=1000 个线程。

假设一个应用中有三个 MySQL 数据库连接池，那么最坏情况下有：

1000×3=3000 个线程创建。

如果数据库采用了分库分表或者读写分离，那么超时带来的影响可想而知。

而 Oracle 采用不同的策略——每个 ClassLoader 一个 watchdog 线程(类似于 MySQL 的 timer)。每个 Statement 一个 Task，而线程是在 watchdog 需要取消时去触发的，即 watchdog 发现该 Statement 需要 cancel 时，调用其某个方法，该方法快速创建线程并运行。

也就是，说假设我们有 500 个连接池，每个连接执行 1 个 Statement，最坏的情况下会创建：

1+500×1=501 个线程。

假设一个应用中有三个 MySQL 库，那么最坏情况下有：

1 + 500×3=1501 个线程创建。

12.1.4 连接池使用的一些建议

一是要注意网络阻塞/不稳定时的级联效应（比如笔者写的 ssdb-client 在网络出现故障如网络不可用时，会设置一个时间，在这个时间内的请求全部 timemout)。连接池内部应该根据当前网络的状态（比如超时次数太多），对于一定时间内的（如 100ms）全部 timeout，根本不进行 await(maxWait)，即有熔断和快速失败机制。

还有就是当前等待连接池的人数，比如现在等待 1000 个，那么接下来的等待是没有意义的，这样还会造成滚雪球效应。

二是等待超时应该尽可能小点（除非很必要）。即使返回错误页，也比等待并阻塞强。DBCP 比较容易出的问题就是设置超时时间太长，造成大量 TIMED_WAIT 和线程阻塞，而且像滚雪球，一旦出问题很难立即恢复，但可以通过上文中的方案解决。

本文通过 DBCP 2 解释了在使用连接池时要注意的一些参数配置，不管使用什么连接池组件，其原理基本类似。如果你对性能追求没有那么极致，则使用 DBCP 2 已经够用了。如果你对性能要求非常高，可以用阿里开源的 Druid，或者号称性能最好的 Java 数据库连接池 HikariCP 等，笔者在实际项目中使用较多的是 Druid。

12.2 HttpClient 连接池

在实际项目中，我们使用 HttpClient 进行 HTTP 服务访问，笔者用过 HttpClient 3.x、4.x。目前最新版本是 5.x（官方文档说 5.x 未来会加入 HTTP/2 作为主要传输协议）。而 3.x、4.x 和 5.x API 是完全不兼容的，用起来很痛苦。4.3.x 和 4.2.x API 也有一些升级，但是向后兼容。本文将介绍 4.5.2、4.2.3、3.1 这三个版本的连接池配置和一些问题。

12.2.1 HttpClient 4.5.2 配置

```
static PoolingHttpClientConnectionManager manager = null;
static CloseableHttpClient httpClient = null;
public static synchronized CloseableHttpClient getHttpClient() {
    if (httpClient == null) {
        //注册访问协议相关的 Socket 工厂
        Registry<ConnectionSocketFactory> socketFactoryRegistry =
             RegistryBuilder.<ConnectionSocketFactory>create()
               .register("http", PlainConnectionSocketFactory.INSTANCE)
               .register("https",
                  SSLConnectionSocketFactory.getSystemSocketFactory())
               .build();
        //HttpConnection 工厂：配置写请求/解析响应处理器
        HttpConnectionFactory<HttpRoute,    ManagedHttpClientConnection>
connFactory = new ManagedHttpClientConnectionFactory(
               DefaultHttpRequestWriterFactory.INSTANCE,
               DefaultHttpResponseParserFactory.INSTANCE);
        //DNS 解析器
        DnsResolver dnsResolver = SystemDefaultDnsResolver.INSTANCE;
        //创建池化连接管理器
      manager = new PoolingHttpClientConnectionManager (
                     socketFactoryRegistry, connFactory, dnsResolver);
```

```
//默认为 Socket 配置
SocketConfig defaultSocketConfig = SocketConfig.custom()
        .setTcpNoDelay(true).build();
manager.setDefaultSocketConfig(defaultSocketConfig);

manager.setMaxTotal(300);//设置整个连接池的最大连接数
//每个路由的默认最大连接，每个路由实际最大连接数默认为
//DefaultMaxPerRoute 控制，而 MaxTotal 是控制整个池子最大数
//设置过小无法支持大并发(ConnectionPoolTimeoutException:
//Timeout waiting for connection from pool)，路由是对 maxTotal 的细分
manager.setDefaultMaxPerRoute(200);//每个路由最大连接数
//在从连接池获取连接时，连接不活跃多长时间后需要进行一次验证，默认为 2s
manager.setValidateAfterInactivity(5 * 1000);

//默认请求配置
RequestConfig defaultRequestConfig = RequestConfig.custom()
        .setConnectTimeout(2 * 1000)//设置连接超时时间，2s
        .setSocketTimeout(5 * 1000)//设置等待数据超时时间，5s
        .setConnectionRequestTimeout(2000)
        //设置从连接池获取连接的等待超时时间
        .build();

//创建 HttpClient
httpClient = HttpClients.custom()
        .setConnectionManager(manager)
        .setConnectionManagerShared(false) //连接池不是共享模式
        .evictIdleConnections(60, TimeUnit.SECONDS)
        //定期回收空闲连接
        .evictExpiredConnections() //定期回收过期连接
        .setConnectionTimeToLive(60, TimeUnit.SECONDS)
        //连接存活时间，如果不设置，则根据长连接信息决定
        .setDefaultRequestConfig(defaultRequestConfig)
        //设置默认请求配置
        .setConnectionReuseStrategy(DefaultConnectionReuseStrategy
.INSTANCE) //连接重用策略，即是否能 keepAlive
        .setKeepAliveStrategy(DefaultConnectionKeepAliveStrategy
.INSTANCE) //长连接配置，即获取长连接生产多长时间
        .setRetryHandler(new DefaultHttpRequestRetryHandler(0,
false)) //设置重试次数，默认是 3 次；当前是禁用掉（根据需要开启）
        .build();

//JVM 停止或重启时，关闭连接池释放掉连接（跟数据库连接池类似）
Runtime.getRuntime().addShutdownHook(new Thread() {
```

```
                @Override
                public void run() {
                    try {
                        httpClient.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
        return httpClient;
    }
```

通过 maxtTotal 和 defaultMaxPerRoute 限制，每个路由（IP+PORT）最多创建 defaultMaxPerRoute 个连接，且所有路由总连接数不超过 maxTotal，即 maxtTotal 是整个池子的大小，defaultMaxPerRoute 是每个路由的大小。比如 maxtTotal=300、defaultMaxPerRoute=200，连接到 http://jd.com 时，到这个主机的并发最多只有 200 而不是 400。连接到 http://jd.com 和 http://qq.com 时，到每个主机的并发最多只有 200，但总的并发连接数为 300。

也可以通过如下方法为某个路由单独设置其连接数大小。

```
manager.setMaxPerRoute(new HttpRoute(new HttpHost("jd.com", 80)), 100);
```

setConnectionManager 方法用于配置 HttpClient 使用的连接池，而 setConnectionManagerShared 方法用于配置此连接池是否在多个 HttpClient 之间共享（默认为 false），如果共享的话，那么如 IdleConnectionEvictor 就不能每个 HttpClient 一个，而只需要定义一个即可。

通过 evictIdleConnections 和 evictExpiredConnections 方法配置一个后台线程定期释放过期连接和空闲连接。HttpClientBuilder 将创建 IdleConnectionEvictor 并定期进行过期，如果连接池是共享的，多个 HttpClient 共用一个连接池，则这两个配置无效。

```
if (!this.connManagerShared) {//只有连接池是非共享模式时
    ……
    final HttpClientConnectionManager cm = connManagerCopy;
    //创建释放连接定时器，其测试周期使用 maxIdleTime，如果不配，则默认为 5s
    if (evictExpiredConnections || evictIdleConnections) {
        final IdleConnectionEvictor connectionEvictor =
            new IdleConnectionEvictor(
                cm,
                maxIdleTime > 0 ? maxIdleTime : 10,
```

```
                maxIdleTimeUnit != null ?
                        maxIdleTimeUnit : TimeUnit.SECONDS);
        //添加 HttpClient#close 回调，当关闭 HttpClient 时，自动关闭该定时器
        closeablesCopy.add(new Closeable() {
            @Override
            public void close() throws IOException {
                connectionEvictor.shutdown();
            }
        });
        connectionEvictor.start();
    }
    //添加 HttpClient#close 回调，当关闭 HttpClient 时自动关闭连接池
    closeablesCopy.add(new Closeable() {
        @Override
        public void close() throws IOException {
            cm.shutdown();
        }
    });
}
```

IdleConnectionEvictor 核心代码如下。

```
while (!Thread.currentThread().isInterrupted()) {
    Thread.sleep(sleepTimeMs);
    connectionManager.closeExpiredConnections();
    if (maxIdleTimeMs > 0) {
        connectionManager.closeIdleConnections(maxIdleTimeMs,
TimeUnit.MILLISECONDS);
    }
}
```

在进行释放过期连接和空闲连接时，IdleConnectionEvictor 会周期性调用 closeExpiredConnections 和 closeIdleConnections 这两个方法，但它们的实现是通过一把大的锁锁住了整个连接池，然后进行遍历。另外，建议只启用 closeExpiredConnections，这需要 HTTP 服务生产者在返回响应中包含超时时间“Keep-Alive: timeout=time”，这样就不需要使用 closeIdleConnections 进行过期空闲连接了。

使用 HttpClient 时，要按照如下模式使用。

```
HttpResponse response = null;
try {
    HttpGet get = new HttpGet("http://item.jd.com/2381431.html");
    response = getHttpClient().execute(get);
    if(response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
```

```
            EntityUtils.consume(response.getEntity());
            //error
        } else {
            String result = EntityUtils.toString(response.getEntity());
            //ok
        }
    } catch (Exception e) {
        if(response != null) {
            EntityUtils.consume(response.getEntity());
        }
    }
```

要使用 EntityUtils.consume(response.getEntity())或者 EntityUtils.toString (response.getEntity())消费响应体，不推荐 HttpEntity#getContent#close 方法来释放连接，处理不好异常将导致连接不释放，也不推荐使用 CloseableHttpResponse#close 关闭连接，它将直接关闭 Socket，导致长连接不能复用。

须要注意的是：

- 在开启长连接时才是真正的连接池，如果是短连接，则只是作为一个信号量来限制总请求数，连接并没有实现复用。
- JVM在停止或重启时，记得关闭连接池释放连接。
- HttpClient是线程安全的，不要每次使用创建一个。
- 如果连接池配置得比较大，则可以考虑创建多个HttpClient实例，而不是使用一个HttpClient实例。
- 使用连接池时，要尽快消费响应体并释放连接到连接池，不要保持太久。

12.2.2 HttpClient 连接池源码分析

连接池实现代码（MainClientExec#execute）。

```
//1. 发送请求并接收响应
response = requestExecutor.execute(request, managedConn, context);

//2. 判断响应是否是长连接，即可复用
if (reuseStrategy.keepAlive(response, context)) {
    //3. 获取长连接超时周期（如果服务器端没设置，则认为永不过期）
    final long duration = keepAliveStrategy.getKeepAliveDuration(response,
context);
    //4. 设置过期周期，并标记连接为可复用
    connHolder.setValidFor(duration, TimeUnit.MILLISECONDS);
```

```
        connHolder.markReusable();
    } else {//标记连接不可复用
        connHolder.markNonReusable();
    }
```

接下来，看看哪些连接可以复用（DefaultConnectionReuseStrategy#keepAlive）。

- 如果有响应头“Transfer-Encoding”且不是“chunked”，则不能复用。
- 如果没有响应头“Transfer-Encoding”，如果响应状态码为 status >= HttpStatus.SC_OK && status != HttpStatus.SC_NO_CONTENT && status != HttpStatus.SC_NOT_MODIFIED && status != HttpStatus.SC_RESET_CONTENT，如果没有响应头“Content-Length”，则不能复用，如果响应头为“Content-Length”>= 0，则可复用，否则不能复用。
- 响应头“Connection”或“Proxy-Connection”为“Close”不能复用。
- HTTP/1.1即使没有响应头“Connection:Keep-Alive”，默认就可复用；而HTTP/1.0必须有响应头“Connection:Keep-Alive”才能复用。

过期时间是通过获取响应头“Keep-Alive: timeout=time”中的 time 实现的，默认实现为 DefaultConnectionKeepAliveStrategy。

当我们使用 EntityUtils 消费内容时（如用 consume 方法），会将连接释放回连接池，这也是为什么让大家获取到响应后尽快消费的原因，在释放连接时有如下代码。

```
if (reusable) {//1. 如果可以复用，则释放到池中
    this.manager.releaseConnection(
         this.managedConn, this.state, this.validDuration, this.tunit);
} else {//2. 不可以复用
    //2.1 关闭物理连接（即 Socket）
    this.managedConn.close();
    //2.2 连接还是会释放到池中（按照之前说的就是一个信号量的作用，
    //且物理连接每次都关闭）
    this.manager.releaseConnection(
          this.managedConn, null, 0, TimeUnit.MILLISECONDS);
```

12.2.3 HttpClient 4.2.3 配置

如下是 HttpClient 4.2.3 配置。

```
public static synchronized HttpClient getHttpClient() {
    if (httpClient == null) {
        // 设置组件参数, HTTP 协议的版本,1.1/1.0/0.9
        HttpParams params = new BasicHttpParams();
```

```
            //设置连接超时时间
            Integer CONNECTION_TIMEOUT = 2 * 1000;  //设置请求超时为 2s
            Integer SO_TIMEOUT = 2 * 1000;          //设置等待数据超时时间为 5s
            Long CONN_MANAGER_TIMEOUT = 1L * 1000;
    //定义了当从 ClientConnectionManager 中检索 ManagedClientConnection
    //实例时使用的毫秒级的超时时间
            params.setIntParameter(CoreConnectionPNames.CONNECTION_TIMEOUT,
CONNECTION_TIMEOUT);
            params.setIntParameter(CoreConnectionPNames.SO_TIMEOUT,
SO_TIMEOUT);
            //在提交请求之前 测试连接是否可用

params.setBooleanParameter(CoreConnectionPNames.STALE_CONNECTION_CHECK,
true);
            //这个参数期望得到一个 java.lang.Long 类型的值。如果这个参数没有被设置，
            //则连接请求就不会超时（无限大的超时时间）
            params.setLongParameter(ClientPNames.CONN_MANAGER_TIMEOUT,
CONN_MANAGER_TIMEOUT);
            PoolingClientConnectionManager conMgr = new PoolingClientConnection
Manager();
            conMgr.setMaxTotal(300);//设置最大连接数
            //是每个路由的默认最大连接
            conMgr.setDefaultMaxPerRoute(100);
            //设置访问协议
            conMgr.getSchemeRegistry().register(new Scheme("http", 80,
PlainSocketFactory. getSocketFactory()));
            conMgr.getSchemeRegistry().register(new Scheme("https", 443,
SSLSocketFactory.getSocketFactory()));
            httpClient = new DefaultHttpClient(conMgr, params);
            httpClient.setHttpRequestRetryHandler(new
DefaultHttpRequestRetryHandler(0, false));
            httpClient.setKeepAliveStrategy(new
DefaultConnectionKeepAliveStrategy());
            manager = conMgr;
        }
        return httpClient;
    }
```

HttpClient 4.2.3 默认没有提供 IdleConnectionEvictor，需要自己实现。HttpClient 3.x 就不介绍了，因为其使用 synchronized+wait+notifyAll。现在存在两个问题，量大时 synchronized 慢且 notifyAll 可能造成线程饥饿。httpclient 4.x 使用 ReentrantLock（默认非公平）+ Condition（每个线程一个）。在笔者机器上（jdk1.6.0_43）测试结果锁的优势明显比较大。

1x synchronized {} with 32 threads took 2.621 seconds

1x Lock.lock()/unlock() with 32 threads took 1.951 seconds

1x synchronized {} with 64 threads took 2.621 seconds

1x Lock.lock()/unlock() with 64 threads took 1.983 seconds

12.2.4 问题示例

此处有一个库存项目的例子，HttpClient 一天并发量在 1500 万左右，峰值每秒 7 万。在之前使用过程中，一直存在大量的如下提示。

```
org.apache.http.conn.ConnectionPoolTimeoutException: Timeout waiting f
or connection from pool
    atorg.apache.http.impl.conn.PoolingClientConnectionManager.leaseConnec
tion(PoolingClientConnectionManager.java:232)
    atorg.apache.http.impl.conn.PoolingClientConnectionManager$1.getConnec
tion(PoolingClientConnectionManager.java:199)
    at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultR
equestDirector.java:456)
```

通过 jstack 查看线程，会发现如下代码。

```
"pool-21-thread-3" prio=10 tid=0x00007f6b7c002800 nid=0x40ff waiting on
condition [0x00007f6b37020000]
    java.lang.Thread.State: TIMED_WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000000f97918b8> (a java.util.concurrent.locks.
AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.parkUntil(LockSupport.java:239)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.
awaitUntil(AbstractQueuedSynchronizer.java:2072)
    ……
    at org.apache.http.pool.PoolEntryFuture.get(PoolEntryFuture.java:100)
    at org.apache.http.impl.conn.PoolingClientConnectionManager.
leaseConnection(PoolingClientConnectionManager.java:212)
```

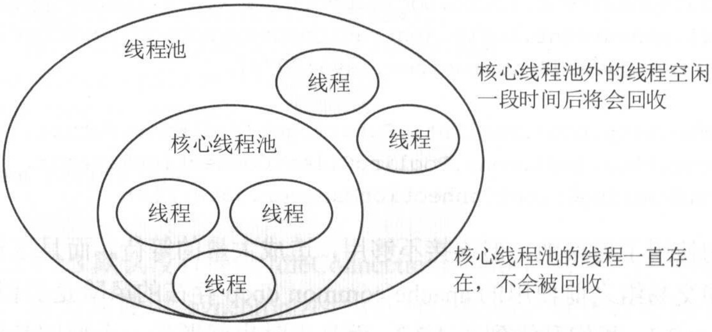
原因是因为使用了连接池，但连接不够用，造成大量的等待。而且这种等待都有滚雪球的效应（和交易组之前使用的 apache common dbcp 存在的风险是类似的）。当时使用的是 HttpClient 3.1，我们升级到了 4.2.3。而且当时出问题的一个原因是使用人员对参数不了解，随意设置其值，不出现问题则好，出现问题很难排查到原因。因此，建议大家按照本文说的进行参数设置。

还有一个问题是RequestConfig默认开启了压缩支持(contentCompressionEnabled)，其会注册 RequestAcceptEncoding（自动添加请求头 Accept-Encoding: gzip,deflate）和 ResponseContentEncoding（自动解压并移除响应头 Content-Length、Content-Encoding、Content-MD5），所以通过 HttpResponse 获取不到这几个响应头也不要奇怪。如果需要这几个头，则可以写自己的 HttpResponseInterceptor 拦截器进行处理。

12.3 线程池

线程池的目的类似于连接池，通过减少频繁创建和销毁线程来降低性能损耗。每个线程都需要一个内存栈，用于存储如局部变量、操作栈等信息，可以通过-Xss 参数来调整每个线程栈大小（64 位系统默认 1024KB，可以根据实际情况调小，比如 256KB），通过调整该参数可以创建更多的线程，不过 JVM 不能无限制地创建线程，通过使用线程池可以限制创建的线程数，从而保护系统。线程池一般配合队列一起工作，使用线程池限制并发处理任务的数量。然后设置队列的大小，当任务超过队列大小时，通过一定的拒绝策略来处理，这样可以保护系统免受大流量而导致崩溃——只是部分拒绝服务，还是有一部分是可以正常服务的。

线程池一般有核心线程池大小和线程池最大大小配置，当线程池中的线程空闲一段时间时将会被回收，而核心线程池中的线程不会被回收。



多少个线程合适呢？建议根据实际业务情况来压测决定，或者根据利特尔法则来算出一个合理的线程池大小，其定义是，在一个稳定的系统中，长时间观察到的平均用户

数量 L，等于长时间观察到的有效到达速率 λ 与平均每个用户在系统中花费的时间的乘积，即 L= λW。但实际情况是复杂的，如存在处理超时、网络抖动都会导致线程花费时间不一样。因此，还要考虑超时机制、线程隔离机制、快速失败机制等，来保护系统免遭大量请求或异常情况的冲击。

Java 提供了 ExecutorService 的三种实现。

- **ThreadPoolExecutor：**标准线程池。
- **ScheduledThreadPoolExecutor：**支持延迟任务的线程池。
- **ForkJoinPool：**类似于ThreadPoolExecutor，但是使用work-stealing模式，其会为线程池中的每个线程创建一个队列，从而用work-stealing（任务窃取）算法使得线程可以从其他线程队列里窃取任务来执行。即如果自己的任务处理完成了，则可以去忙碌的工作线程那里窃取任务执行。

12.3.1 Java 线程池

使用 Executors 来创建线程池。

1. 创建单线程的线程池。

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
```

等价于

```
return new FinalizableDelegatedExecutorService
    (new ThreadPoolExecutor(1, 1,
                            0L, TimeUnit.MILLISECONDS,
                            new LinkedBlockingQueue<Runnable>()))
```

2. 创建固定数量的线程池。

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
```

等价于

```
return new ThreadPoolExecutor(nThreads, nThreads,
                              0L, TimeUnit.MILLISECONDS,
                              new LinkedBlockingQueue<Runnable>());
```

3. 创建可缓存的线程池，初始大小为 0，线程池最大大小为 Integer.MAX_VALUE。其使用 SynchronousQueue 队列，一个没有数据缓冲的阻塞队列。对其执行 put 操作后必须等待 take 操作消费该数据，反之亦然。该线程池不限制最大大小，如果线程池有空闲

线程则复用，否则会创建一个新线程。如果线程池中的线程空闲 60 秒，则将被回收。该线程默认最大大小为 Integer.MAX_VALUE，请确认必要后再使用该线程池。

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

等价于

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                        60L, TimeUnit.SECONDS,
                        new SynchronousQueue<Runnable>())
```

4. 支持延迟执行的线程池，其使用 DelayedWorkQueue 实现任务延迟。

```
ScheduledExecutorService scheduledExecutorService =
                  Executors. newScheduledThreadPool(10);
```

等价于

```
return new ScheduledThreadPoolExecutor(corePoolSize, Integer.MAX_VALUE,
                             0, NANOSECONDS,
                             new DelayedWorkQueue());
```

5. work-stealing 线程池，默认并行数为 Runtime.*getRuntime*(). availableProcessors()。

```
ExecutorService executorService = Executors.newWorkStealingPool(5);
```

等价于

```
return new ForkJoinPool(parallelism,
                ForkJoinPool.defaultForkJoinWorkerThreadFactory(),
               null, true);
```

接下来，我们来看一下 ThreadPoolExecutor 配置。

- **corePoolSize**：核心线程池大小，线程池维护的线程最小大小，即没有任务处理情况下，线程池可以有多个空闲线程，类似于DBCP中的minIdle。
- **maximumPoolSize**：线程池最大大小，当任务数非常多时，线程池可创建的最大线程数量。
- **keepAliveTime**：线程池中线程的最大空闲时间，存活时间超过该时间的线程会被回收，线程池会一直缩小到corePoolSize大小。
- **workQueue**：线程池使用的任务缓冲队列，包括有界阻塞数组队列ArrayBlockingQueue、有界/无界阻塞链表队列LinkedBlockingQueue、优先级阻塞队列PriorityBlockingQueue、无缓冲区阻塞队列SynchronousQueue。有界阻塞队列须要设置合理的队列大小。

- **threadFactory**：创建线程的工厂，我们可以设置线程的名字、是否是后台线程。
- **rejectedExecutionHandler**：当缓冲队列满后的拒绝策略，包括Abort（直接抛出RejectedExecutionException）、Discard（按照LIFO丢弃）、DiscardOldest（按照LRU丢弃）、CallsRun（主线程执行）。

Spring 也提供了 XML 标签用来方便创建线程池。一个是：

```
<task:executor id="asyncTaskExecutor"
            pool-size="${executor.pool.size}"
            queue-capacity="${executor.queue.capacity}"
            keep-alive="60"/>
```

另一个是：

```
<task:scheduler id="scheduler" pool-size="10"/>
```

6. 线程池终止

线程池不再使用后记得停止掉，可以调用 shutdown 以确保不接受新任务，并等待线程池中任务处理完成后再退出，或调用 shutdownNow 清除未执行任务，并用 Thread.interrupt 停止正在执行的任务。然后调用 awaitTermination 方法等待终止操作执行完成，代码如下。

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        executorService.shutdown();
        executorService.awaitTermination(30, TimeUnit.SECONDS)
    }
});
```

7. 总结

根据任务类型是IO密集型还是CPU密集型、CPU核数，来设置合理的线程池大小、队列大小、拒绝策略，并进行压测和不断调优来决定适合自己场景的参数。

笔者遇到过因为maximumPoolSize设置的过大导致瞬间线程数非常多。还有使用如Executors.newFixedThreadPool时，因没有设置队列大小，默认为Integer.MAX_VALUE，如果有大量任务被缓存到 LinkedBlockingQueue 中等待线程执行，则会出现 GC 慢等问题，造成系统响应慢甚至 OOM。因此，在使用线程池时务必须设置池大小、队列大小并设置相应的拒绝策略（RejectedExecutionHandler）。线程池执行情况下无法捕获堆栈上下文，因此任务要记录相关参数，以方便定位提交任务的源头及定位引起问题的源头。

12.3.2 Tomcat线程池配置

以 Tomcat 8 为例配置如下，配置方式一。

```
<Connector port="8080" acceptCount="100" maxConnections="200"
         minSpareThreads ="10" maxThreads="200"/>
```

- **acceptCount**：请求等待队列大小。当Tomcat没有空闲线程处理连接请求时，新来的连接请求将放入等待队列，默认为100。当队列超过acceptCount后，新连接请求将被拒绝。
- **maxConnections**：Tomcat能处理的最大并发连接数。当超过后还是会接收连接并放入等待队列（acceptCount控制），连接会等待，不能被处理。BIO默认是maxThreads数量。NIO和NIO2默认是10000，ARP默认是8192。
- **minSpareThreads**：线程池最小线程数，默认为10。该配置指定线程池可以维持的空闲线程数量。
- **maxThreads**：线程池最大线程数，默认为200。当线程池空闲一段时间后会释放到只保留minSpareThreads个线程。

举例，假设 maxThreads=100，maxConnections=50，acceptCount=50，假设并发请求为 200，则有 50 个线程并发处理 50 个并发连接，50 个连接进入等待队列，剩余 100 个将被拒绝。也就是说 Tomcat 最大并发线程数是由 maxThreads 和 maxConnections 中最小的一个决定。BIO 场景下 maxConnections 和 maxThreads 是一样的，当我们需要长连接场景时，应使用 NIO 模式，并发连接数是大于线程数的。

配置方式二。

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-" deamon= "true"
        minSpareThreads="25" maxThreads="200" maxIdleTime="60000"
        maxQueueSize= "Integer.MAX_VALUE"
        prestartminSpareThreads="false"/>
<Connector port="8080" executor="tomcatThreadPool"
        executorTerminationTimeoutMillis ="5000"/>
```

此处我们使用了 org.apache.catalina.Executor 实现，其表示一个可在多个 Connector 间共享的线程池，而且有更丰富的配置。

- **namePrefix**：创建的Tomcat线程名字的前缀。
- **deamon**：是否守护线程运行，默认为true。
- **minSpareThreads**：线程池最小线程数，默认为25。
- **maxThreads**：线程池最大线程数，默认为200。

- **maxIdleTime：**空闲线程池的存活时间，默认为60s。当线程空闲超过该时间后，线程将被回收。
- **maxQueueSize：**任务队列最大大小，默认为Integer.MAX_VALUE，建议改小。可以认为是maxConnections。
- **prestartminSpareThreads：**是否在Tomcat启动时就创建minSpareThreads个线程放入线程池，默认为false。
- **executorTerminationTimeoutMillis：**在停止Executor时，等待请求处理线程终止的超时时间。

最后，要根据业务场景和压测来配置合理的线程池大小，配置太大的线程池在并发量较大的情况下会引起请求处理不过来导致响应慢，甚至造成 Tomcat 僵死。

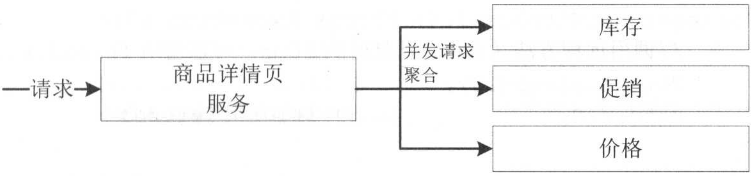
在本书出版时，Docker 容器中使用 Runtime.*getRuntime*(). availableProcessors()获取到的是物理机核数，而不是容器实际使用的核数，这将对性能造成极大影响。所有用到该参数的地方都要记得调整，如 CMS 垃圾回收参数：-XX:ParallelGCThreads 和 -XX:ConcGCThreads，可扫二维码参考《使用 Docker 容器时不要忘记进行 GC 参数审查》。

13 异步并发实战

在做电商系统时，首页、活动页、商品详情页等系统承载了网站的大部分流量，而这些系统的主要职责包括聚合数据拼装模板、热点统计、缓存、下游功能降级开关、托底数据等。其中聚合数据需要调用多个其他服务获取数据、拼装数据/模板，然后返回给前端，聚合数据来源主要有依赖系统/服务、缓存、数据库等。而系统之间的调用可以通过如 HTTP 接口调用（如 HttpClient）、SOA 服务调用（如 dubbo、thrift）等实现。

在 Java 中，如使用 Tomcat，一个请求会分配一个线程进行请求处理，该线程负责获取数据、拼装数据或模板，然后返回给前端。在同步调用获取数据接口的情况下（等待依赖系统返回数据），整个线程是一直被占用并阻塞的。如果有大量的这种请求，则每个请求占用一个线程，但线程一直处于阻塞，降低了系统的吞吐量，这将导致应用的吞吐量下降。我们希望，在调用依赖的服务响应比较慢时，应该让出线程和 CPU 来处理下一个请求，当依赖的服务返回后再分配相应的线程来继续处理。而这应该有更好的解决方案：异步/协程。而 Java 是不支持协程的（虽然有些 Java 框架号称支持，但还是高层 API 的封装），因此，在 Java 中我们可以使用异步来提升吞吐量。目前大部分 Java 开源框架（HttpAsyncClient、Dubbo、Thrift 等）都支持。

另外，应用中一个服务可能会调用多个依赖服务来处理业务，而这些依赖服务是可以同时调用的。如果顺序调用的话需要耗时 100ms，而并发调用只需要 50ms，那么可以使用 Java 并发机制来并发调用依赖服务，从而降低该服务的响应时间。



在开发应用系统过程中，通过异步并发并不能使响应变得更快，更多是为了提升吞吐量、对请求更细粒度控制，或是通过多依赖服务并发调用降低服务响应时间。当一个线程在处理任务时，通过 Fork 多个线程来处理任务并等待这些线程的处理结果，这种应用并不是真正的异步。异步是针对 CPU 和 IO 的，当 IO 没有就绪时要让出 CPU 来处理其他任务，这才是异步。本文不会介绍异步并发实现原理，主要介绍在 Java 应用中如何运用这些技术，而且大多数场景并不是真正的异步化，在 Java 中真正实现异步化是非常困难的事情，如 MySQL JDBC 驱动等很多都是 BIO 设计，大多数情况下说的异步并发是通过线程池模拟实现。

13.1 同步阻塞调用

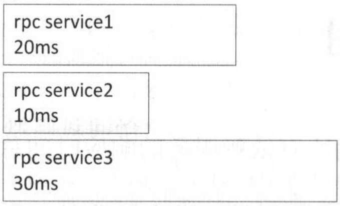
即串行调用，响应时间为所有依赖服务的响应时间总和。

```
public class Test {
   public static void main(String[] args) throws Exception {
       RpcService rpcService = new RpcService();
       HttpService httpService = new HttpService();
       //耗时为 10ms
       Map<String, String> result1 = rpcService.getRpcResult();
       //耗时为 20ms
       Integer result2 = httpService.getHttpResult();
       //总耗时为 30ms
   }
   static class RpcService {
       Map<String, String> getRpcResult() throws Exception {
           //调用远程方法（远程方法耗时约 10ms，可以使用 Thread.sleep 模拟）
       }
   }

   static class HttpService {
```

```
        Integer getHttpResult() throws Exception {
            //调用远程方法（远程方法耗时约 20ms，可以使用 Thread.sleep 模拟）
            Thread.sleep(20);
            return 0;
        }
    }
}
```

13.2 异步 Future

线程池配合 Future 实现，但是阻塞主请求线程，高并发时依然会造成线程数过多、CPU 上下文切换。通过 Future 可以并发发出 *N* 个请求，然后等待最慢的一个返回，总响应时间为最慢的一个请求返回的用时。如下请求如果并发访问，则响应可以在 30ms 后返回。




```
public class Test {
    final static ExecutorService executor =
                                    Executors.newFixedThreadPool(2);
    public static void main(String[] args) {
        RpcService rpcService = new RpcService();
        HttpService httpService = new HttpService();
        Future<Map<String, String>> future1 = null;
        Future<Integer> future2 = null;
        try {
            future1 = executor.submit(() -> rpcService.getRpcResult());
            future2 = executor.submit(() -> httpService.getHttpResult());
            //耗时为 10ms
            Map<String, String> result1 =
                            future1.get(300, TimeUnit.MILLISECONDS);
            //耗时为 20ms
            Integer result2 = future2.get(300, TimeUnit.MILLISECONDS);
```

```
            //总耗时为 20ms
        } catch (Exception e) {
            if (future1 != null) {
                future1.cancel(true);
            }
            if (future2 != null) {
                future2.cancel(true);
            }
            throw new RuntimeException(e);
        }
    }
    static class RpcService {
        Map<String, String> getRpcResult() throws Exception {
            //调用远程方法（远程方法耗时约 10ms，可以使用 Thread.sleep 模拟）
        }
    }
    static class HttpService {
        Integer getHttpResult() throws Exception {
            //调用远程方法（远程方法耗时约 20ms，可以使用 Thread.sleep 模拟）
        }
    }
}
```

13.3 异步 Callback

通过回调机制实现，即首先发出网络请求，当网络返回时回调相关方法，如HttpAsyncClien 使用基于 NIO 的异步 I/O 模型实现，它实现了 Reactor 模式，摒弃阻塞 I/O 模型 one thread per connection，采用线程池分发事件通知，从而有效支撑大量并发连接。这种机制并不能提升性能，而是为了支撑大量并发连接或者提升吞吐量。

```
public class AsyncTest {
    public static HttpAsyncClient httpAsyncClient;
    public static CompletableFuture<String> getHttpData(String url) {
        CompletableFuture asyncFuture = new CompletableFuture();

        HttpAsyncRequestProducer producer =
                        HttpAsyncMethods.create(new HttpPost(url));
        BasicAsyncResponseConsumer consumer =
```

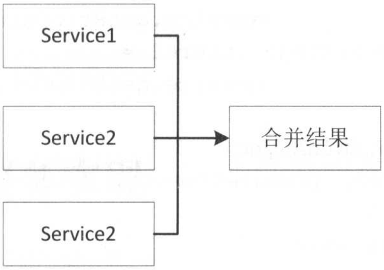
```
                    new BasicAsyncResponseConsumer();

        FutureCallback callback = new FutureCallback<HttpResponse>() {
            public void completed(HttpResponse response) {
                asyncFuture.complete(response);
            }
            public void failed(Exception e) {
                asyncFuture.completeExceptionally(e);
            }
            public void cancelled() {
                asyncFuture.cancel(true);
            }
        };

        httpAsyncClient.execute(producer, consumer, callback);
        return asyncFuture;
    }

    public static void main(String[] args) throws Exception {
        CompletableFuture<String> future =
                        AsyncTest.getHttpData ("http://www.jd.com");
        String result = future.get();
    }
}
```

这种异步实现可以配合 CompletableFuture 实现半异步。

13.4 异步编排 CompletableFuture

JDK 8 CompletableFuture 提供了新的异步编程思路，可以对多个异步处理进行编排，实现更复杂的异步处理。其内部使用 ForkJoinPool 实现异步处理。使用 CompletableFuture 可以把回调方式的实现转变为同步调用实现。CompletableFuture 提供了 50 多个 API，可以满足各种所需场景的异步处理编排，在此列举三个场景。

场景一是三个服务异步并发调用，然后对结果合并处理，不阻塞主线程。




```
public static void test1() throws Exception {
   MyService service = new MyService();
   CompletableFuture<String> future1 =
                             service.getHttpData("http:// www.jd.com");
   CompletableFuture<String> future2 =
                             service.getHttpData("http:// www.jd.com");
   CompletableFuture<String> future3 =
                             service.getHttpData("http://www.jd.com");

   CompletableFuture.allOf(future1, future2, future3)
          .thenApplyAsync((Void) -> {
             //异步处理 future1 future2 future3 结果
          }).exceptionally(e -> {
             //处理异常
          });
}
```

如上方式直接通过 thenApplyAsync 异步处理 future1~3 的结果，不阻塞主线程，内部使用 ForkJoinPool 线程池实现。也可以通过返回一个新的 CompletableFuture 来同步处理结果，即阻塞主线程。

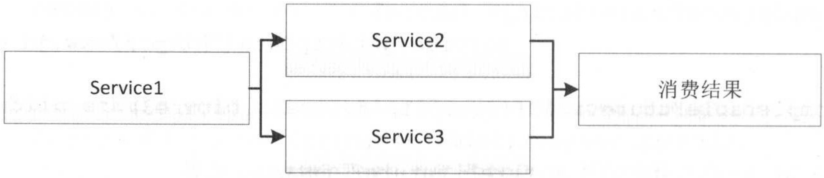
```
CompletableFuture<List> future4 = CompletableFuture
      .allOf(future1, future2, future3)
      .thenApply((Void) -> {
         return Lists.newArrayList(
                              future1.get(), future2.get(), future3.get())
      }).exceptionally(e -> {
         //处理异常
      });
```

场景二是两个服务并发调用，然后消费结果，不阻塞主线程。

```
public static void test2() throws Exception {
   MyService service = new MyService();

   CompletableFuture<String> future1 =
```

```
                                    service.getHttpData("http:// www.jd.com");
    CompletableFuture<String> future2 =
                                    service.getHttpData("http:// www.jd.com");

    future1.thenAcceptBothAsync(
                future2, (future1Result, future2Result) -> {
        //异步处理结果
    }).exceptionally(e -> {
        //异常处理
    });
}
```

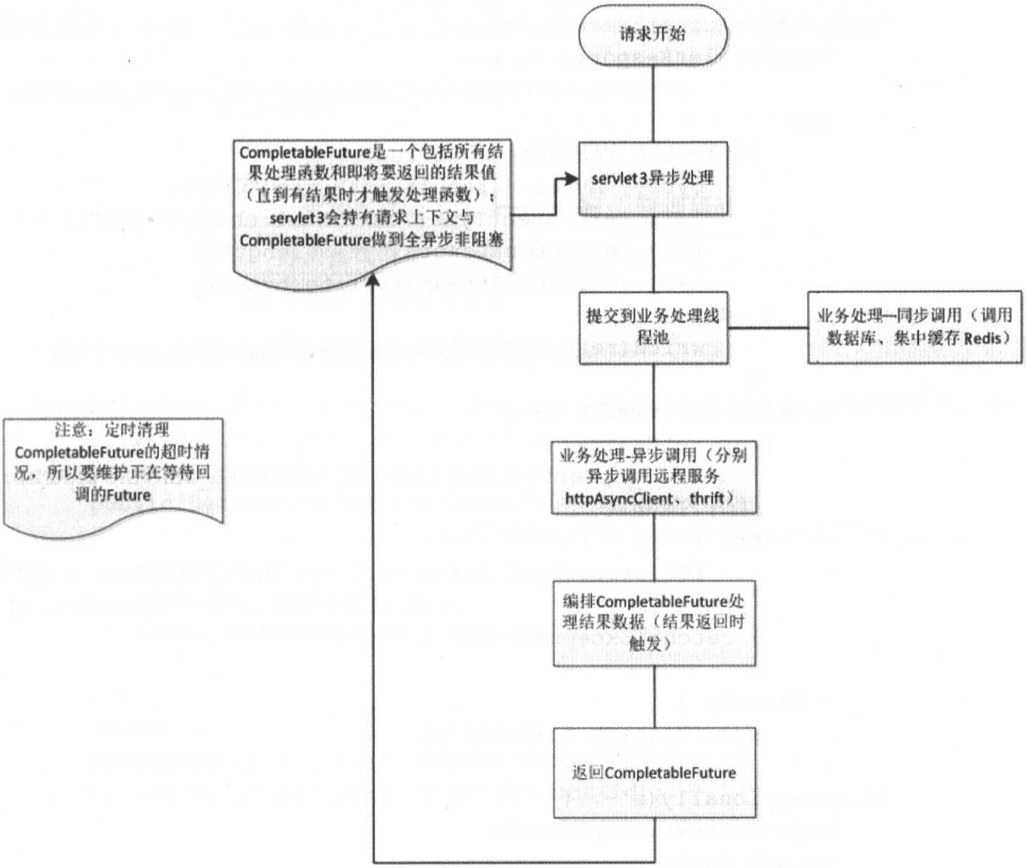
场景三是服务 1 执行完成后，接着并发执行服务 2 和服务 3，然后消费相关结果，不阻塞主线程。




```
public static void test3() throws Exception {
    MyService service = new MyService();

    CompletableFuture<String> future1 =
                                service.getHttpData("http:// service1");
    CompletableFuture<String> future2 =
                        future1.thenApplyAsync((v) -> {
                            return "result from service2";
                        });
    CompletableFuture<String> future3 =
                                service.getHttpData("http:// service3");
    future2.thenCombineAsync(future3, (f2Result, f3Result) -> {
        //处理业务
    }).exceptionally(e -> {
        //处理异常
    });
}
```

13.5 异步 Web 服务实现

借助 Servlet 3、CompletableFuture 实现异步 Web 服务。如下是整个处理流程。



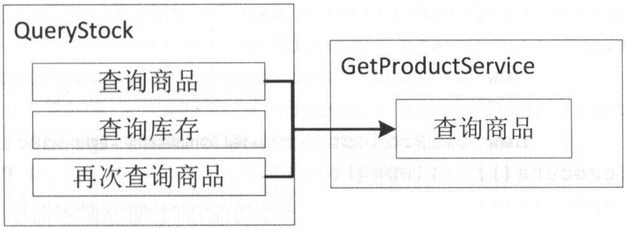
Servlet 容器接收到请求之后，Tomcat 需要先解析请求体，然后通过异步 Servlet 将请求交给异步线程池来完成业务处理，Tomcat 线程释放回容器。通过异步机制可以提升 Tomcat 容器的吞吐量。

```
public void submitFuture(finalHttpServletRequest req, final Callable
```

```
<CompletableFuture> task) throwsException{
        final String uri = req.getRequestURI();
        final Map<String, String[]> params = req.getParameterMap();
        final AsyncContext asyncContext = req.startAsync();
        asyncContext.getRequest().setAttribute("uri", uri);
        asyncContext.getRequest().setAttribute("params", params);
        asyncContext.setTimeout(asyncTimeoutInSeconds * 1000);
        if(asyncListener != null) {
            asyncContext.addListener(asyncListener);
        }
        CompletableFuture future = task.call();
        future.thenAccept(result -> {
            HttpServletResponse resp =
                    (HttpServletResponse)asyncContext.getResponse();
            try {
                if(result instanceof String) {
                    byte[] bytes = result.getBytes("GBK");
                    resp.setContentType("text/html;charset=gbk");
                    resp.setContentLength(bytes.length);
                    resp.getOutputStream().write(bytes);
                } else {
                    write(resp,JSONUtils.toJSON(result));
                }
            } catch (Throwable e) {
                resp.setStatus(
                        HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
                //程序内部错误
                try {
                    LOG.error("get infoerror, uri : {},  params : {}",
                                    uri,JSONUtils.toJSON(params), e);
                } catch (Exception ex) {
                }
            } finally {
                asyncContext.complete();
            }
        }).exceptionally(e -> {
            asyncContext.complete();
            return null;
        });
    }
```

13.6 请求缓存

在一个查询库存的服务中，因为一些特殊原因对同一个商品查询了多次，即一次用户请求需要重复调用多次商品接口。我们一般的做法是将 GetProductService 包装一层 JVM 缓存，不过，使用 Hystrix 后，我们还有另一种请求级别的缓存实现。



GetProductServiceCommand 实现代码如下。

```
public class GetProductServiceCommand extends HystrixCommand<String> {
    private ProductService productService;
    private Long id;
    public GetProductServiceCommand(
                                ProductService productService, Long id) {
        super(setter());
        this.id = id;
        this.productService = productService;
    }
    ......
    @Override
    protected String run() throws Exception {
        return productService.getProduct(id);
    }

    @Override
    protected String getCacheKey() {
        return "product-" + id;
    }
}
```

此处需要实现 getCacheKey 方法，指定缓存 key。

需要使用 withRequestCacheEnabled(true)配置开启请求缓存支持。

```
HystrixCommandProperties.Setter commandProperties =
    HystrixCommandProperties.Setter()
        .withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrategy.THREAD)
        .withRequestCacheEnabled(true)//默认 true
```

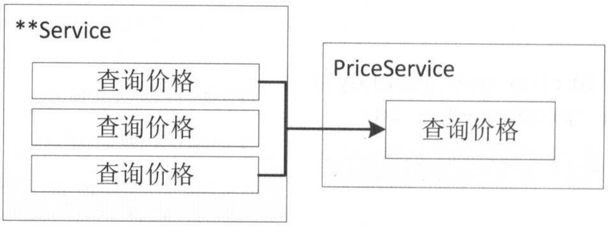
业务代码调用实现如下。

```
HystrixRequestContext context = HystrixRequestContext. initializeContext();
try {
    ProductService productService = new ProductService();
    GetProductServiceCommand command1 =
                    new GetProductServiceCommand (productService, 1L);
    GetProductServiceCommand command2 =
                    new GetProductServiceCommand (productService, 1L);
    command1.execute();
    command2.execute();
    Assert.assertFalse(command1.isResponseFromCache());
    Assert.assertTrue(command2.isResponseFromCache());
} finally {
    context.shutdown();
}
```

Hystrix 使用了 ThreadLocal HystrixRequestContext 实现，并在异步线程执行之前注入 ThreadLocal HystrixRequestContext 实现多个线程共享，从而实现请求级别的响应缓存。

下面看一下如何用 CompletableFuture 实现批量查询。

我们有个服务需要多次查询价格，而价格服务提供了单个查询和批量查询接口。一种方式是我们在客户端多线程查询，然后聚合。另一种方式是调用批量查询接口（一些服务器端实现其实是串行的，这种情况建议使用客户端多线程查询，而不是服务器端提供的支持）。在调用批量接口时，我们需要限制每次批量的大小，从而减少阻塞时间。



使用 CompletableFuture 实现客户端多线程批量查询。

```
List<CompletableFuture<Double>> futures = Lists.newArrayList();
for(Long id : ids) {
    futures.add(CompletableFuture.supplyAsync(
                    () -> {return priceService. getPrice(id);}));
}
CompletableFuture.allOf(
                    futures.toArray(new CompletableFuture[0])).get();
```

因为我们知道价格有批量接口，也可以在客户端调用批量接口实现。但是，我们不能一次批量查询太多价格数据，服务器端限定我们每次最多查询 10 个。因此，我们需要对 id 进行分区，以实现批量查询。

```
List<CompletableFuture<List<Double>>> futures = Lists.newArrayList();
List<List<Long>> pages = Lists.partition(ids, 10);
for(List<Long> page : pages) {
    futures.add(CompletableFuture.supplyAsync(() -> {
        return priceService.getPrices(page);
    }));
}
CompletableFuture.allOf(
futures.toArray(new CompletableFuture[0])).get();
```

13.7 请求合并

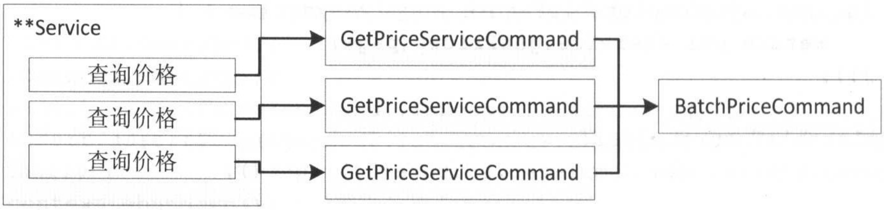
CompletableFuture 必须提前构造好批量查询，而 Hystrix 支持将多个单个请求转换为单个批量请求，即可以按照单个命令来请求。但是，实际是以批量请求模式执行。

Hystrix 请求合并业务代码如下。

```
HystrixRequestContext context =
                    HystrixRequestContext.initializeContext();
try {
    PriceService priceService = new PriceService();
    GetPriceServiceCommand command1 =
                    new GetPriceServiceCommand (priceService, 1L);
    GetPriceServiceCommand command2 =
                    new GetPriceServiceCommand (priceService, 2L);
```

```
        GetPriceServiceCommand command3 =
                              new GetPriceServiceCommand (priceService, 3L);

        Future<Double> f1 = command1.queue();
        Future<Double> f2 = command2.queue();
        Future<Double> f3 =command3.queue();

        f1.get();
        f2.get();
        f3.get();
    } finally {
        context.shutdown();
    }
```

可以看到，业务代码还是单个价格查询。Hystrix 内部会将多个查询进行合并后批量查询，此处需要先使用 queue 而不能直接使用 execute 方法调用。



当我们调用 GetPriceServiceCommand 时，最终会将请求合并，然后交由 BatchPriceCommand 执行批量查询。

GetPriceServiceCommand 实现

```
public class GetPriceServiceCommand
                extends HystrixCollapser <List<Double>, Double, Long> {
    private PriceService priceService;
    private Long id;
    public GetPriceServiceCommand(PriceService priceService, Long id) {
        super(setter());
        this.priceService = priceService;
        this.id = id;
    }

    private static HystrixCollapser.Setter setter() {
        return HystrixCollapser.Setter
                .withCollapserKey(HystrixCollapserKey.Factory.asKey("pri
ce"))
                .andCollapserPropertiesDefaults(HystrixCollapserProperti
```

```
es.Setter()
                    .withMaxRequestsInBatch(2)
                    .withTimerDelayInMilliseconds(5)
                    .withRequestCacheEnabled(true))
                .andScope(Scope.REQUEST);
    }

    @Override
    public Long getRequestArgument() {
        return id;
    }

    @Override
    protected  HystrixCommand<List<Double>>  createCommand(Collection
<CollapsedRequest<Double, Long>> requests) {
        return new BatchPriceCommand(priceService, requests);
    }

    @Override
    protected  void  mapResponseToRequests(List<Double>  batchResponse,
Collection<CollapsedRequest<Double, Long>> requests) {
        final AtomicInteger count = new AtomicInteger(0);
        requests.forEach((request) -> {
            request.setResponse(
                        batchResponse.get(count.getAndIncrement()));
        });
    }
}
```

HystrixCollapser.Setter 配置

- **collapserKey**：配置全局唯一标识服务合并的名称，类似于HystrixCommandKey。如果不配置，则默认是简单类名，通过该名称进行请求合并。
- **collapserPropertiesDefaults**：maxRequestsInBatch配置每个请求合并允许的最大请求数，如果请求多于此配置会分多批次执行，默认为Integer.MAX_VALUE。timerDelayInMilliseconds配置在批处理执行之前的等待超时时间，默认为10ms。requestCacheEnabled如果跨多请求进行请求合并，则必须开启，开启后可以消除重复请求，默认为true。
- **scope**：请求合并范围，默认为Scope.REQUEST，即当前请求上下文。如果配置为Scope.GLOBAL，则表示全局，即可以跨越多个请求上下文进行请求合并。

如果需要GLOBAL，则记得开启requestCacheEnabled。建议只使用REQUEST范围，如果非要使用GLOBAL，那么请给出合理的理由。

timerDelayInMilliseconds 是请求合并时执行的延迟时间，如果请求合并数量正好等于 maxRequestsInBatch，那么就不需要等待而立即执行。但是，如果请求数量<maxRequestsInBatch，那么请求会在该超时时间后才执行，其使用线程池的scheduleAtFixedRate(r, listener.getIntervalTimeInMilliseconds(), listener.getIntervalTimeInMilliseconds(), TimeUnit.MILLISECONDS)实现。

HystrixCollapser 的实现方法如下。

- **getRequestArgument**：返回请求参数，如果有多个参数需要包装为一个，参数会被封装为CollapsedRequest。
- **createCommand**：创建可批量执行的Command，当HystrixCollapser对请求进行合并后达到maxRequestsInBatch时或timerDelayInMilliseconds超时后，就会创建批处理命令，如示例中的BatchPriceCommand。
- **mapResponseToRequests**：将执行结果映射到请求中，从而单个请求就可以获得结果了。
- **shardRequests**：如果想把不同的请求分到不同的分组进行请求合并，可以使用该命令，如HashTag应用。比如一个商品有商品基本信息(p:id:)、商品介绍(d:id:)、颜色尺码(c:id:)等标签，我们存储时如不采用HashTag将会导致这些数据不会存储到一个分片，而是分散到多个分片。这样获取时需要从多个分片获取数据进行合并，无法进行mget。如果有了HashTag，那么可以使用“::”中间的数据做分片逻辑，这样id一样的将会分到一个分片。shardRequests可以按照HashTag类似机制实现。

BatchPriceCommand 实现

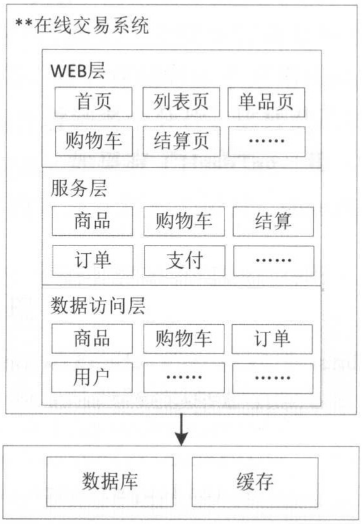
```
class BatchPriceCommand extends HystrixCommand<List<Double>> {
    private PriceService priceService;
    private Collection<CollapsedRequest<Double, Long>> requests;
    public    BatchPriceCommand(PriceService    priceService,    Collection<
CollapsedRequest<Double, Long>> requests) {
        super(setter());
        this.priceService = priceService;
        this.requests = requests;
    }
    private static Setter setter() {
        return Setter.withGroupKey(
```

```
                HystrixCommandGroupKey.Factory.asKey("price"));
    }

    @Override
    protected List<Double> run() throws Exception {
        List<Long> ids = requests.stream()
                .map(req -> {return req.getArgument();})
                .collect(Collectors.toList());
        return priceService.getPrices(ids);
    }
}
```

普通 Command 实现是，将要合并请求的请求进行合并，然后调用批量查询接口，从而将多个单次查询合并为一个批量查询。

14 如何扩容

对于一个发展初期的系统来说，不太确定商业模型到底行不行，最好的办法是按照最小可行产品方法进行产品验证，因此，刚开始的功能会比较少，是一个大的单体应用，一般按照三层架构进行设计开发，使用单数据库，缓存也是可选组件，而应用系统和数据库也很可能部署在同一台物理机上，如下图所示。



对于这样一个系统，随着产品使用的用户越来越多，网站的流量会增加，最终单台服务器无法处理那么大的流量，此时就需要用分而治之的思想来解决问题。

第一步是尝试通过简单扩容来解决。

第二步，如果简单扩容搞不定，就需要水平拆分和垂直拆分数据/应用来提升系统的伸缩性，即通过扩容提升系统负载能力。

第三步，如果通过水平拆分/垂直拆分还是搞不定，那就需要根据现有系统特性，从架构层面进行重构甚至是重新设计，即推倒重来。

对于系统设计，理想的情况下应支持线性扩容和弹性扩容，即在系统瓶颈时，只需要增加机器就可以解决系统瓶颈，如降低延迟提升吞吐量，从而实现扩容需求。

如果你想扩容，则支持水平/垂直伸缩是前提。在进行拆分时，一定要清楚知道自己的目的是什么，拆分后带来的问题如何解决，拆分后如果没有得到任何收益就不要为了拆而拆，即不要过度拆分，要适合自己的业务。本章主要从应用和数据层面讲解如何按照业务和功能进行应用或数据层面的拆分。

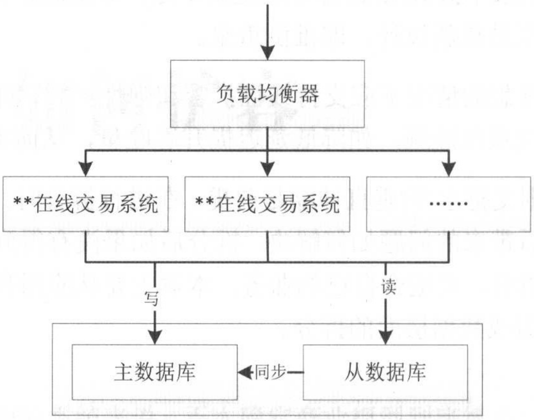
14.1 单体应用垂直扩容

有些时候，如果能通过硬件快速解决，而且成本不高，应该首先通过硬件扩容来解决问题。硬件扩容包括升级现有服务器，比如 CPU 由原来的 32 核升级到 64 核；内存从 64GB 升级到 256GB（有的缓存服务器 CPU 利用率很低，但是内存不够用，就通过扩容内存来提升单机容量）；磁盘扩容，比如系统有大量的随机读写，因此把 HDD 换成 SSD，还有将原来单机 1TB 扩容为 2TB；原来硬盘做了 RAID 10，现在直接拆为裸盘使用，通过架构层面提升数据可靠性。此外，核心数据库可以使用 PCIe SSD 或 NVMe SSD，千兆网卡可以升级为万兆网卡。不管怎么扩容，单机总会是瓶颈，而分布式技术是提升系统扩容能力的更好方法。

14.2 单体应用水平扩容

单体系统水平扩容是通过部署更多的镜像来实现的。如下图所示，原来通过一个系

统实例对外提供服务，通过扩容到更多实例后，用户访问时不可能提供多个域名/IP 入口，应该提供统一入口，此时就需要负载均衡机制来实现。



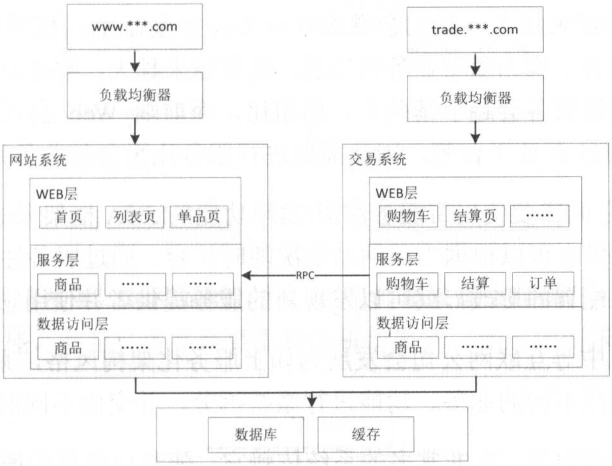
如果用户会话数据分散在应用系统，就需要在负载均衡器开启会话黏滞特性。

如果数据库的瓶颈是读造成的，则此时可以通过主从数据库架构将读的流量分散到更多的从服务器上，写数据时写到主数据库，读数据时读取从数据库。

经过单体应用的垂直/水平扩容，如果系统还是有瓶颈，则此时只有通过拆分应用来解决。

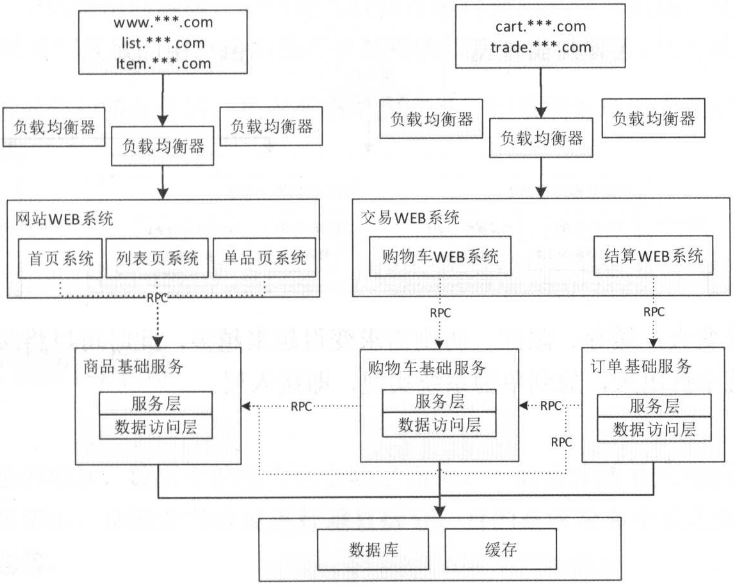
14.3 应用拆分

对于单体应用来说，随着业务量的增加，一个大系统就会有很多人维护，这就造成修改代码会出现冲突，上线必须大家一起上线，而且风险较大，导致需求实现速度缓慢。因此单体应用发展到一定地步时，会按照业务进行拆分。



如上图所示，我们按照业务将一个大系统拆分为多个子系统，比如网站系统和交易系统。拆分时要进行业务代码解耦，将功能分离到不同系统上。拆分后系统之间是物理隔离的，应用层面原来是直接进程内方法调用，现在需要改成远程方法调用，比如通过WebService、RMI 等。

通过拆分，可以由两个团队分别维护网站和交易系统，相互之间的更新是不冲突的。但是目前也存在一些问题，比如，我们使用 RMI 机制，需要使用方维护一个服务方 IP 列表。因此下一个方向是 SOA 化，如下图所示。

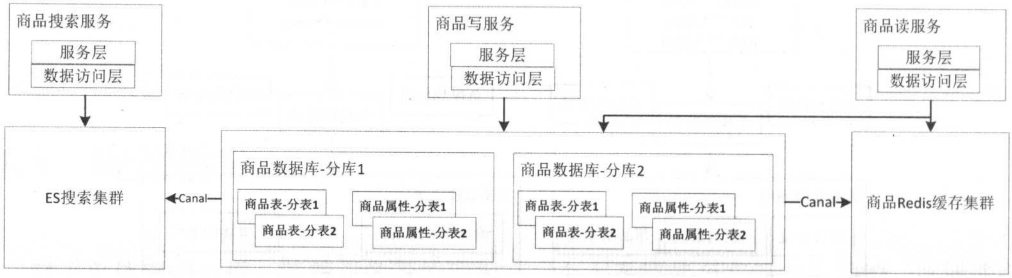


随着系统流量越来越大，我们会继续在业务拆分基础上，按照功能域拆分为前端Web系统和基础服务。因为随着业务的发展，流量越来越大，解决方案越来越复杂。像商品、购物车、结算服务会趋于基础化、通用化，而前端 Web 会有各种各样的版本和需求，如PC/APP/H5/开放平台等，因此需要进行服务化平台与业务系统的拆分。

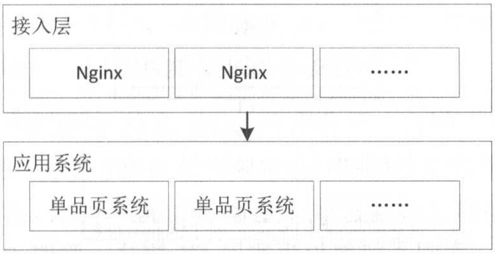
拆分后，系统之间需要使用带服务注册/发现功能的SOA框架来进行交互，如Dubbo。服务化后，服务提供者可以根据当前网站状况随时扩容。通过服务注册中心，服务消费者不需要进行任何配置的更改，就可以发现新的服务提供者并使用它。

一般情况下，中等互联网公司会发展为如上服务化架构风格，系统之间通过 SOA 服务进行互动，按照不同的业务、功能进行系统拆分，并交由不同的团队维护。

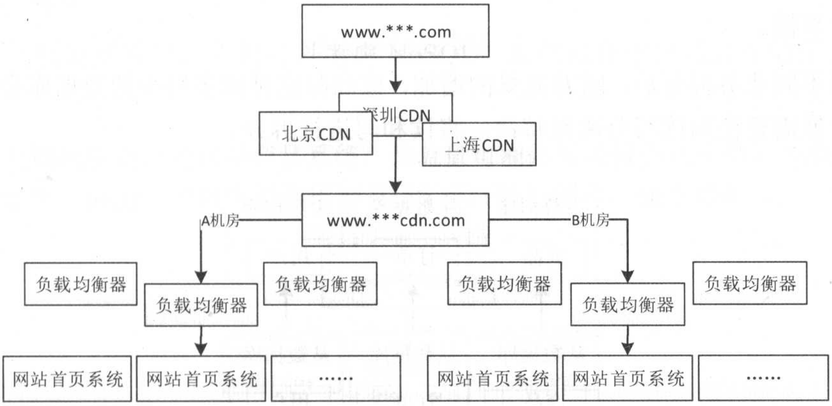
像商品这种基础服务，有非常多的系统依赖它。随着访问量的增加，尤其像单个读/单个写/条件查询这类访问，会因为某一种操作出现异常造成其他操作不可用。因此，我们需要把这些操作进行拆分，拆分到不同的服务中，从而使写出现问题时不会影响到读。另外，因为进行了系统拆分，主数据库向缓存/ES同步时会有一定的延迟，如果需要强一致性的读，那么直接读主库吧。但是，不是所有的系统都需要读主库，要做出限制。随着应用部署数量的增多，数据库连接也会成为瓶颈，一般会通过主从架构提升连接数。也可以使用 MyCat/Corbar 这种数据库中间件提升连接数。所有应用只调用读/写服务中间件，由读/写服务中间件访问数据库，减少整体的连接数。然后通过 MQ 异构数据，从而不访问有瓶颈的数据库。



随着流量变大，缓存、限流、防刷需求变得越来越多，此时可以将缓存/限流/防刷从各应用系统中拆出来，放到单独系统实现，即接入层。



另外，随着网站发展，对网站的性能、可用性要求越来越高，对于前端页面型应用需要引进 CDN 功能，并且业务系统要支持多机房多活，如下图所示。

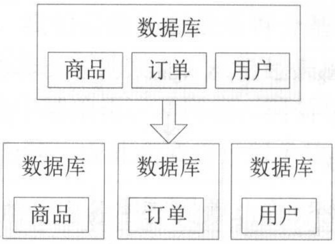


当其中一个机房出问题时，应该能比较快速地切换到另一个机房。使用 BIND 可以根据用户 IP 将不同区域的用户路由到离他最近的机房来提供服务，从而减少访问延迟。

通过应用拆分和服务化后，扩容变得更加容易，如系统处理能力跟不上，只需要增加服务器即可。

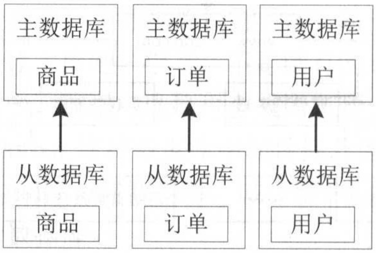
14.4 数据库拆分

随着流量的增加，数据库的压力也会随之而来，一般会伴随着应用拆分进行数据库拆分。如下图所示，按照业务维度进行垂直拆分，目的是解决多个表之间的 IO 竞争、单机容量问题等。

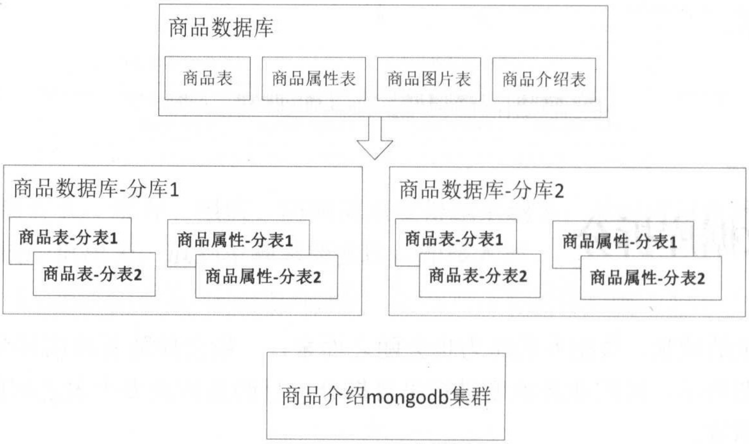


拆分后会出现，原来可以进行单库 join 查询，现在不可以了，需要解决跨库 join，还要解决分布式事务等问题。跨库 join 可以考虑通过如全局表、ES 搜索等异构数据机制来实现。数据库垂直拆分中还存在一种宽表拆多个小表的场景，不过一般在设计时就会做这件事情。

按照不同业务拆分后，随着流量的增加，像商品这种读多写少的数据库会遇到读瓶颈，此时就需要使用读写分离来解决，将读和写进行拆分。



随着流量和数据量的增加，单库单表会遇到容量和磁盘/带宽 IO 瓶颈，单表会随着数据量增长出现性能瓶颈，此时就需要分库、分表，或者分库分表。

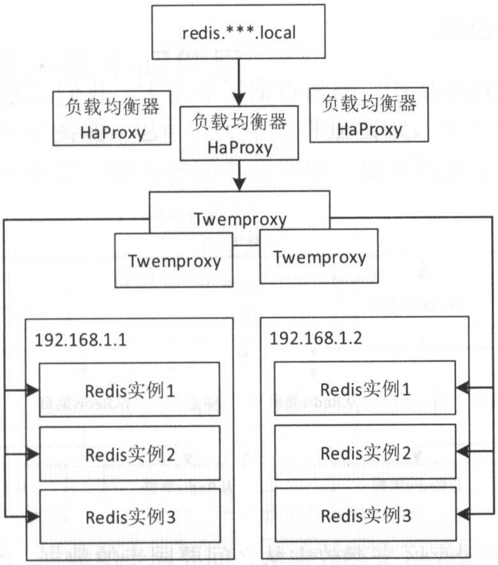


分库分表是一种水平数据拆分，会按照如 ID、用户、时间等维度进行数据拆分，拆分算法可以是取模、哈希、区间或者使用数据路由表等。

这也导致了前文中说的跨库/跨表 join、排序分页、自增 ID、分布式事务等问题。对于跨库/跨表 join 和排序分页，可以对所有表进行扫描然后做聚合，或者生成全局表、进行查询维度的数据异构（比如，订单库按照查询维度异构出商家订单库、用户订单库），再或者将数据同步到 ES 搜索。自增 ID 问题可以通过不同表、不同自增步长或分布式 ID 生成器解决。而分布式事务可以考虑事务表、补偿机制（执行/回滚）、TCC 模式（预占/确认/取消）、Sagas 模式（拆分事务+补偿机制）等，业务应尽量设计为最终一致性，而不是强一致性。

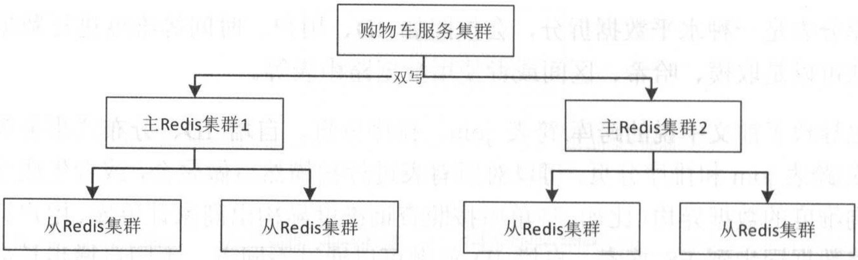
对于一些特殊数据，我们可以考虑 NoSQL，如商品介绍很适合存储在 mongodb 集群中。

对于互联网应用，尤其是商品系统，读流量可能是写流量的几十倍，而单个商品的查询会非常多，此时，可以考虑使用如 Redis 进行数据缓存，如下图所示。



部署多个 Redis 实例，通过 Twemproxy 并使用一致性哈希算法进行分片，先通过 HaProxy 进行 Twemproxy 的负载均衡，然后通过内网域名进行访问。

还有如购物车数据，是用户维度数据，我们完全可以全量存储到 KV 存储中，如使用 Redis 进行存储。为了数据的安全性，我们采用了双写架构，如下图所示。

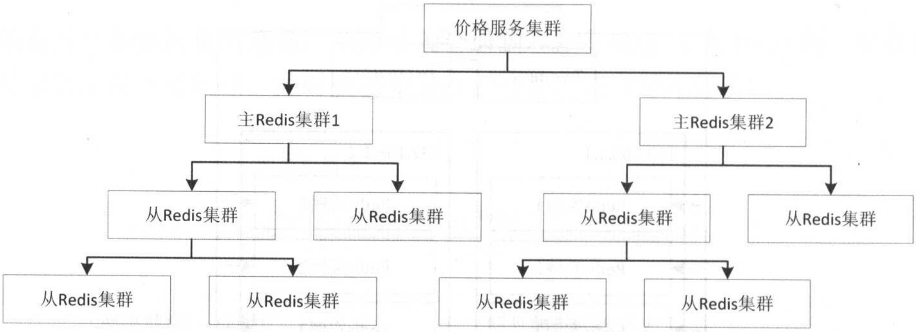


最简单的办法是在多个集群间通过主从来解决，不过主从切换比较麻烦，当主从断开后需要全量更新时恢复较慢。

也可以使用程序双写，实现逻辑比较简单且切换方便。程序双写可以是程序同步双写，写失败其中一个就都失败。这种方式性能差，不适合多机房同步写，也不适合同步写多个集群。

还可以使用异步双写，首先把变更发布到数据总线（如通过 MQ 实现），然后订阅数据总线变更，异步写其他集群。这种方式的优点是性能好，缺点是异步同步有一定的时延，数据一致性差一些，应考虑使用一致性哈希把用户调度到同一个集群，防止用户刷新多次看到不一样的数据。

实时价格类似于购物车架构，因为查询量非常大，我们会通过挂更多的从来扩展读的能力，如下图所示。



Redis 使用内存复制缓存区来存放主从之间要同步的数据。当主从断开时间较长时，复制缓冲区达到阈值，此时旧缓存数据会被丢弃，此时断开的主从进行同步时将会全量复制。Redis 也没有提供类似于 mysql binlog 的机制。

到此应用拆分和数据库拆分就介绍完了。应用扩容可以通过部署更多的应用实例来解决，无法部署更多的实例时，就需要考虑系统拆分或者重新架构。而数据库扩容首先

是硬件层面，然后按照业务进行垂直拆分，接着进行水平拆分，最后根据流量场景进行读写分离，还可以将读流量分流到 NoSQL 上。

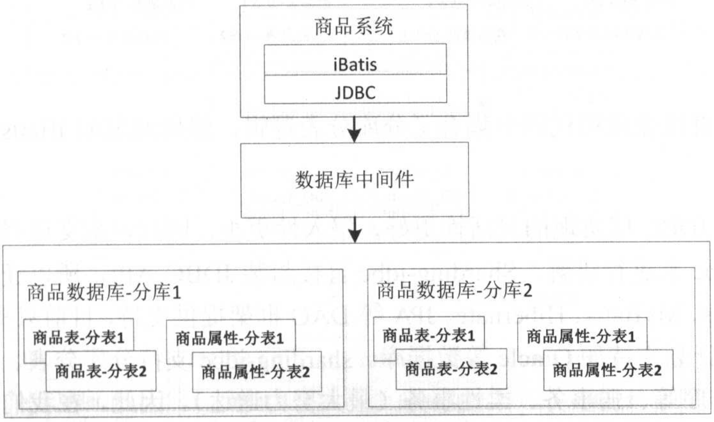
14.5 数据库分库分表示例

数据库分库分表后就会涉及如何写入和读取数据的问题，应用开发人员主要关心如下几个问题。

- 是否需要在应用层做改造来支持分库分表，即是在应用层进行支持，还是通过中间件层呢？
- 如果需要应用层做支持，那么分库分表的算法是什么？
- 分库分表后，join是否支持，排序分页是否支持，事务是否支持。

14.5.1 应用层还是中间件层

分库分表可以在应用层实现，也可以在中间件层实现，中间件层实现的好处是对应用透明，应用就像查单库单表一样去查询中间件层，如下图所示。

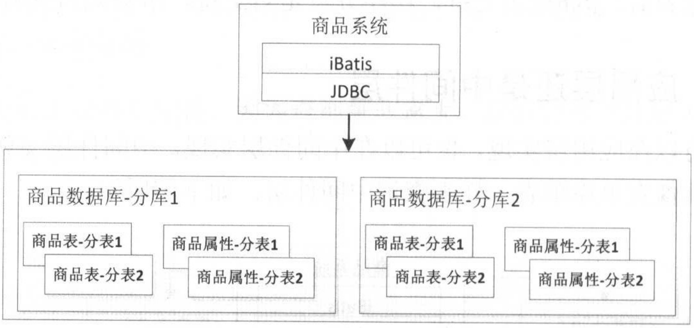


使用数据库中间件层还有一个好处是可以支持多种编程语言，而不受限于特定的语言。使用数据库中间件层可以减少应用的总数据库连接数，从而避免因为应用过多导致数据库连接不够用。缺点是除了维护中间件外，还要考虑中间件的 HA/负载均衡等，增

加了部署和维护的困难，因此，还是要看当前阶段有没有必要使用中间件和有没有人维护该中间件。

目前开源的数据库中间件有基于 MySQL-Proxy 开发的奇虎 360 的 Atlas、阿里的 Cobar、基于 Cobar 开发的 Mycat 等。京东内部也有很多分库分表实现，还有如 JProxy 分布式数据库实现，截止本书出版前暂未开源。Atlas 只支持分表或分库（sharding 版本）、读写分离等，不支持跨库分表（如分 3 个库每个库 3 张表），sharding 版本不支持跨库操作（跨库事务/跨库 join 等）。Cobar 支持分库不支持分表（如每个库 3 个表），不支持跨库 join/分页/排序等。Mycat 支持分库分表、读写分离、跨库弱事务支持，对跨库 join 等有限支持（内存聚合）。这些中间件目前主要支持 MySQL，但 MyCat 也提供了对 Oracle 等数据库的支持。

而应用层可以在 JDBC 驱动层、DAO 框架层，如 iBatis/Mybatis/Hibernate/JPA 上完成。如当当的 sharding-jdbc 是 JDBC 驱动层实现，而阿里的 cobar-client 是基于 DAO 框架 iBatis 实现，如下图所示。



应用系统直接在应用代码中耦合了分库分表逻辑，然后通过如 iBatis/JDBC 直接分库分表实现。

相对来说 JDBC 层实现的灵活性更好，侵入性更少，因此，本文选择了开源的当当的 Sharding-jdbc 来进行讲解。Sharding-jdbc 直接封装 JDBC API，所以迁移成本很低，可以对如 iBatis、MyBatis、Hibernate、JPA 等 DAO 框架提供支持，目前只提供了 MySQL 的支持，未来计划支持如 Oracle 等数据库。sharding-jdbc 支持分库分表、读写分离、跨库 join/分页/排序等、弱事务、柔性事务（最大努力送达）。因此，在我们的场景中需要使用的分库分表/弱事务功能它都有。

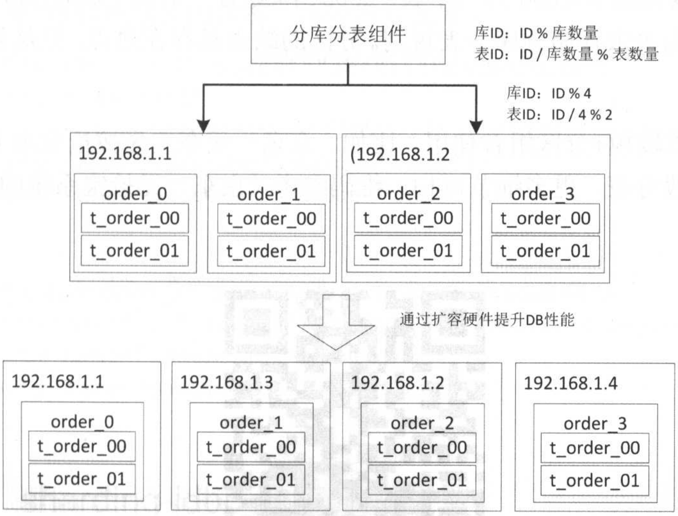
14.5.2 分库分表策略

分库分表策略是指按照什么算法或规则进行存储，它会影响数据的写入和读取，比如，按照订单 ID 分库分表，那么就很难按照客户维度进行订单查询。因此，在进行分库分表时需要慎重考虑使用什么策略。常见的策略有取模、分区、路由表等。

1. 取模

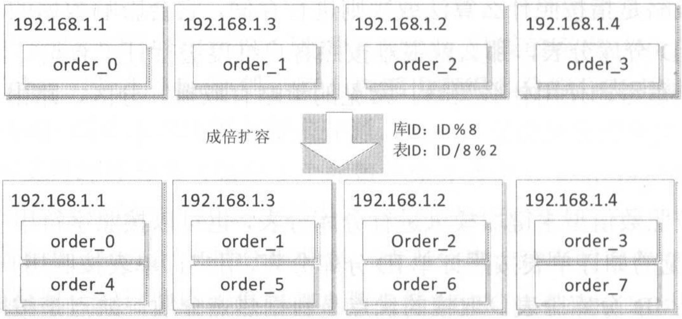
我们可以按照数值型主键取模来进行分库分表，也可以按照字符串主键哈希取模进行分库分表，常见的如订单表按照订单 ID 分库分表，用户订单表按照用户 ID 分库分表，产品表按照产品 ID 分库分表。取模的优点是数据热点分散，缺点是按照非主键维度进行查询时需要跨库/跨表查询，扩容需要建立新集群并进行数据迁移。如果想减少扩容时带来的麻烦，可以在初期规划时冗余足够数量的分库分表，比如一年规划只需要分 2 个库 4 个表，可以冗余设计为 4 个库 8 个表，0-1 库在机器 1，2-3 库在机器 2，如果遇到性能问题时可以把 1、3 库移到新的机器上。如果遇到容量问题，则可以按照如下步骤进行扩容。

每台物理机上有两个数据库实例，当遇到数据库性能瓶颈时首先可以通过升级硬件解决，如 HDD 换成 SATA SSD、SATA SSD 换成 PCIe SSD 或 NVMe SSD；升级硬件之后，瓶颈可能是磁盘空间或者网卡带宽。如果还是不能解决性能问题，接着通过扩容物理机来解决性能瓶颈。



当通过扩容物理机无法解决性能问题或者当单表容量遇到瓶颈，可以进行成倍扩容，

4 个库扩容为 8 个库，如下图所示。



成倍扩容后的数据迁移可以这样实现，先挂数据库主从（order_4-->order_0），当数据库主从同步完成后，停应用写数据库并等待一段时间以保证主从同步完成，接着更新分库分表规则并启动应用进行写库，最后删除各个库的冗余数据即可。

分库数量不是越多越好，可以参考“第 12 章 连接池线程池详解”相关章节，分库太多会导致消耗更多的数据库连接，并且应用会创建更多的线程。这种情况下数据代理中间件会是更好的选择。

2. 分区

可按照时间分区、范围分区进行分库分表，时间分区规则如一个月一个表、一年一个库。范围分区规则如 0~2000 万一个表，2000~4000 万一个表。如果分区规则很复杂，则可以有一个路由表来存储分库分表规则。分区的缺点是存在热点，但是易于水平扩展，能避免数据迁移。

另外，也可以取模+分区组合使用。比如，京东一元夺宝先按抢宝项 Hash 分库，然后按抢宝期区间段分表，更多细节可扫二维码参考《京东一元抢宝系统的数据库架构优化》。

14.5.3 使用 sharding-jdbc 分库分表

在数据库设计起初一般都是单库单表设计，随着数据量的增长将带来存储容量和写/读性能瓶颈问题。如果是容量问题，则可以通过分库到多台机器解决。而引起写/读问题的主要原因是记录太多（几千万到一亿）、列数太多、索引太多、查询太复杂等引发单表出现性能问题。出现性能问题要从很多维度去分析，如果经过分析得以解决，则我们可以继续单库分表，如果单库分表确实有问题，则要进行分库分表解决。比如，电商系统的商品数据库，就会存在这种问题。为了演示方便，我们将商品数据库分为 2 个库，每个库 2 个表，使用 MySQL 数据库。

1. 数据库 DDL

创建 2 个库，每个库 2 个表，即 *N* 为 0、1，*M* 为 0、1，然后执行如下脚本。

```
CREATE DATABASE IF NOT EXISTS product_N;
CREATE TABLE product_M(
  id bigint primary key,
  title varchar(255),
  last_modified datetime
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2. 数据源配置

使用 DBCP 2 配置 2 个 DataSource，abstractDataSource 把公共部分抽取为父 Bean，可参考“第 12 章 连接池线程池详解”部分进行配置。

```
<bean id="dataSource_0" parent="abstractDataSource">
    <property name="url"
              value="jdbc:mysql://192.168.1.2:3306/product _0"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>

<bean id="dataSource_1" parent="abstractDataSource">
    <property name="url" value="jdbc:mysql://192.168.1.3:3306/ product_1"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

14.5.4 sharding-jdbc 分库分表配置

本文使用 sharding-jdbc 1.3.2 依赖。

```xml
<dependency>
    <groupId>com.dangdang</groupId>
    <artifactId>sharding-jdbc-core</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>com.dangdang</groupId>
    <artifactId>sharding-jdbc-transaction</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>com.dangdang</groupId>
    <artifactId>sharding-jdbc-config-spring</artifactId>
    <version>1.3.2</version>
</dependency>
```

如下配置可实现分 2 个库，每个库分 2 个表。

```xml
<!-- 分库规则 -->
<rdb:strategy id="dataSourceStrategy"
         sharding-columns="id"
         algorithm-expression=
                "dataSource_${Math.floorMod(id.longValue(),2L)}"/>
<!-- 分表规则 -->
<rdb:strategy id="productTableStrategy"
         sharding-columns="id"
         algorithm-expression=
            "product_${Math.floorMod(Math.floorDiv(id.longValue(),2L
),2L)}"/>

<!-- 分库分表数据源 -->
<rdb:data-source id="shardingDataSource">
    <!-- 使用的真实数据源 -->
    <rdb:sharding-rule data-sources="dataSource_0,dataSource_1">
        <rdb:table-rules>
            <!-- 分表规则：分库策略、分表策略、逻辑表名、实际表名-->
            <rdb:table-rule
                    database-strategy="dataSourceStrategy"
                    table-strategy="productTableStrategy"
                    logic-table="product"
                    actual-tables="product_${0..1}"/>
        </rdb:table-rules>
    </rdb:sharding-rule>
</rdb:data-source>
```

分库/分表策略：使用 sharding-columns 指定分库分表键，algorithm-expression 指定分库分表策略，我们按照 ID 分了两个库，每个库两张表。算法为：库 ID = id % 库数量，表 ID = id / 库数量 % 单库表数量。另一种算法为：库 ID = id % 表总数量 / 单库表数量，表 ID = id % 表总数量。

分库分表数据源：配置分库数据源，其会按照分库策略（table-rule/ database-strategy）选择使用哪一个数据源。然后使用 table-strategy 配置分表策略来选择使用哪一张表。logic-table 是逻辑表名，写 SQL 时使用这个标识，然后会根据分表策略和 actual-tables 决定真实表名。

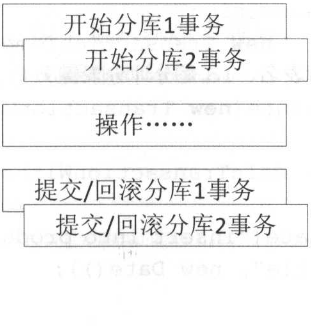
sharding-jdbc 的分库分表算法是独立的，即分库可以使用一套规则，分表可以使用一套规则，如订单库按照商家 ID 分库，然后每个库按照订单 ID 分表。如果你的分库分表策略太复杂，则可以使用 algorithm-class 指定 SingleKeyDatabaseShardingAlgorithm/MultipleKeysDatabaseShardingAlgorithm、SingleKeyTableShardingAlgorithm/MultipleKeysTableShardingAlgorithm 分库分表实现算法，其支持单个键/多个组合键作为分片键。分库分表算法支持如=、BETWEEN、IN 等多维度实现。

1. 事务管理器配置

配置弱事务管理器在大多数场景够用了。

```
<!-- 事务管理器，此处使用弱事务 -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.
                                          DataSourceTransactionManager">
    <property name="dataSource" ref="shardingDataSource"/>
</bean>
```

此处我们使用了弱事务机制，如下图所示，事务不是原子的，可能提交分库 1 事务后，提交分库 2 事务失败，造成跨库事务不一致。可以考虑 sharding-jdbc 提供的柔性事务实现。



柔性事务目前支持最大努力送达，未来计划支持TCC（Try-Confirm-Cancel）。最大努力送达是当事务失败后通过最大努力反复尝试送达操作实现，是在假定数据库操作一定可以成功的前提下进行的，保证数据最终的一致性。其适用场景是幂等性操作，如根据主键删除数据、带主键地插入数据、更新记录最后状态（如商品上下架操作）。

Sharding-JDBC 的最大努力送达型柔性事务分为同步送达和异步送达两种，同步送达不需要 ZooKeeper 和 elastic-job，内置在柔性事务模块中。但是在有些场景下，事务需要经过一段时间才能准备完毕，则可通过异步送达，异步送达比较复杂，是对柔性事务的最终补偿，不能和应用程序部署在一起，需要额外地通过 elastic-job 实现。异步送达是对同步送达的有效补充，但即使不配置异步送达，同步送达机制也可以正常使用。最大努力送达型事务也可能出现错误，即无论如何补偿都不能正确提交。为了避免反复尝试带来的系统开销，同步送达和异步送达均可配置最大重试次数，超过最大重试次数的事务将进入失败列表，需要定期进行人工干预。具体使用请参考 sharding-jdbc 官方文档。

在一般场景中，只要保证单库事务能工作即可，跨库通过一些机制保证最终一致性即可，在高并发高可用的场景下不应该采用强一致模型。

2. 代码逻辑

在实际使用时，通过 JDBC 模板和编程式完成事务开发，通过 AOP 机制配置事务。

```
//获取分库分表数据源
DataSource shardingDataSource =
              (DataSource) ctx.getBean ("shardingDataSource");
//创建 JdbcTemplate
JdbcTemplate jdbcTemplate = new JdbcTemplate(shardingDataSource);
//获取事务管理器
AbstractPlatformTransactionManager transactionManager =
      (AbstractPlatformTransactionManager)ctx.getBean("transactionManager");
//创建事务模板
TransactionTemplate transactionTemplate =
                      new TransactionTemplate (transactionManager);
//执行 SQL（product 是逻辑表名、id 是分库分表键）
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
   @Override
   protected void doInTransactionWithoutResult(TransactionStatus transactionStatus) {
       jdbcTemplate.update("insert into product(id,title,last_modified) values (?, ?, ?)", 1L, "title", new Date());
```

```
    }
});
```

整体使用和非分库分表没什么区别，在实际执行时，逻辑表名 product 会被替换为如 product_1 这种实际表名，即实际 SQL 会是如下样子：

```
INSERT INTO product_1 (id, title, last_modified) VALUES (?, ?, ?)
```

14.5.5 使用 sharding-jdbc 读写分离

随着数据库读访问量的增长，主库不能承受更多的读访问，此时，可以通过给主库挂从库，然后把读访问分流到从库来减少主库的压力。sharding-jdbc 通过简单的配置就可以支持读写分离。

读写分离数据源配置

通过如下配置就可以实现读写分离，即配置一个主库和两个从库。

```
<rdb:master-slave-data-source id="dataSource_0"
     master-data-source-ref ="dataSource_master_0"
slave-data-sources-ref="dataSource_slave_0,dataSource_slave_1"/>
```

使用如上配置读请求会通过路由到达从库，但是，假设刚刚写入数据，此时立即读的话可能读不到，因为 MySQL 默认使用异步复制，复制是有一定延迟的。因此要想在写完后立即读数据，可以通过 Hint 机制强制读取主库。

```
HintManager.getInstance().setMasterRouteOnly();
jdbcTemplate.queryForList("select id, title from product where id=?", 1L);
```

Sharding-JDBC 的读写分离为了最大限度避免由于同步延迟而产生强制读取主库的场景，在更新方面做了优化，在一个请求线程中，只要存在对数据库的更新操作，则在此操作之后的任何对数据库的访问都会自动通过路由达到主库。因此，在写后读的场景中不需要使用 HintManager，只有在读场景下，需要强制读主库时，才使用 HintManager 强制通过路由到达主库。

通过数据库主从分离读写，但不是从库越多越好，当从库同步遇到瓶颈，或者通过从库无法满足查询需求时，应该选择数据异构。

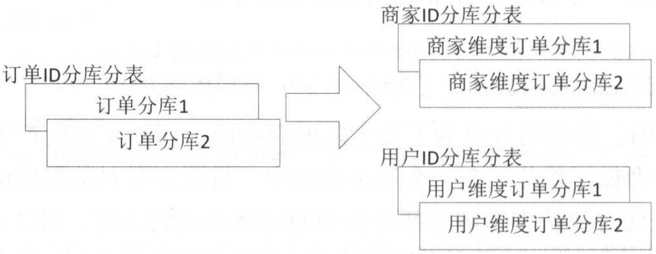
14.6 数据异构

分库分表后将带来很多问题，如跨库 join、非分库分表维度的条件查询、分页排序等。前面我们提到了可以扫描全部表通过内存聚合、数据异构（全局表、ES 搜索、异构表）等来实现。数据异构主要按照不同查询维度建立表结构，这样就可以按照这种不同维度进行查询。数据异构有查询维度异构、聚合数据异构等。

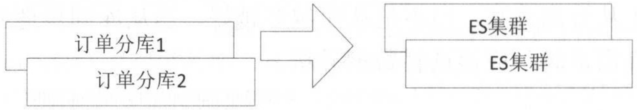
在数据量和访问量双高时使用数据异构是非常有效的，但增加了架构的复杂度。异构时可以通过订阅 MQ 或者 binlog 并解析实现。

14.6.1 查询维度异构

比如对于订单库，当对其分库分表后，如果想按照商家维度或者按照用户维度进行查询，那么是非常困难的，因此可以通过异构数据库来解决这个问题。可以采用下图的架构。



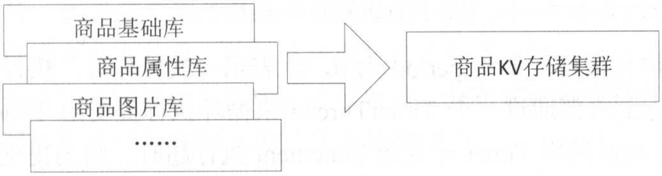
或者采用下图的



异构数据主要存储数据之间的关系，然后通过查询源库查询实际数据。不过，有时可以通过数据冗余存储来减少源库查询量或者提升查询性能。

14.6.2 聚合据异构

商品详情页中一般包括商品基本信息、商品属性、商品图片，在前端展示商品详情页时，是按照商品 ID 维度进行查询，并且需要查询 3 个甚至更多的库才能查到所有展示数据。此时，如果其中一个库不稳定，就会导致商品详情页出现问题，因此，我们把数据聚合后异构存储到 KV 存储集群（如存储 JSON），这样只需要一次查询就能得到所有的展示数据。这种方式也需要系统有了一定的数据量和访问量时再考虑。京东商品详情页就是采用这种异构机制。



具体实现请参考第 15 章中基于 Canal 实现数据异构部分。

14.7 任务系统扩容

在开发系统时，有时需要在特定的时间点执行一些任务，或者周期性地执行一些任务。比如，每天凌晨删除过期的垃圾消息、每天凌晨进行报表统计、每天凌晨进行数据结转，或者每隔 10 分钟处理一次超时未支付的订单、每隔 10 秒删除过期的活动等。对于一般单实例任务，使用如 Thread、Timer、ScheduledExecutor、Quartz 单机版就足够了，如果需要高可用或分布式版本，则可以选择 Quartz 集群版、tbschedule、elastic-job 等。

14.7.1 简单任务

在一般情况下，我们使用 Thread 就能满足需求，如第 15 章中使用 EventPublishThread 线程抓取任务并交给 Disruptor 处理。一般 Thread 的使用方法如下。

```
while(true) {
    //任务处理
    //休眠等待
}
```

即使用 Thread，一般都是死循环抓取并处理任务，如果没有任务，则可以休眠一下，然后继续尝试抓取任务，为了保证任务能及时被处理，休眠时间非常短，一般为几毫秒到几秒。比如要获取任务表中状态为未处理的任务并进行处理，处理成功后将状态更新为已处理，则可以使用如上介绍的 Thread 方式。

如果需要周期性地执行任务，则可以使用 Timer。

```
Timer#schedule(TimerTask task, long delay, long period)
//周期性地执行 TimerTask，首次执行时间为（当前时间+delay）
Timer#schedule(TimerTask task, Date firstTime, long period)
//周期性地执行 TimerTask，首次执行时间为 firstTime
```

period 的单位为毫秒，如果 period 为 0，则表示一次性任务，执行完成后将从任务队列被移除。Timer 内部通过一个 TimerThread 来循环执行提交给 Timer 的任务，比如 MySQL JDBC 驱动就使用 Timer 来处理 Statement 执行超时。因为提交给 Timer 的任务是被单个 TimerThread 处理的，因此任务是串行处理的，前一个任务延迟了将会影响到后续所有任务，当然也可以启动多个 Timer 来实现。但是，如果任务处理不是密集型的，那么将造成线程休眠，从而被浪费，因此，如果需要更灵活的周期性任务处理，则可以使用 ScheduledExecutorService，它基于线程池实现，任务可以被线程池中的一个线程处理，从而实现线程的复用。

```
    ScheduledExecutorService#scheduleAtFixedRate(Runnable command, long
initial Delay, long period, TimeUnit unit) //固定速率执行周期性任务
    ScheduledExecutorService#scheduleWithFixedDelay(Runnable command, long
initialDelay, long period, TimeUnit unit) //固定延迟执行周期性任务
```

其中，initialDelay 指定初次执行延迟，period 指定周期，unit 指定延迟和周期的时间单位。scheduleAtFixedRate 是用固定速率执行周期性任务，即相对于上一次任务开始时间往后推 period 时间后执行下一次任务。假设上一次任务开始时间为 10:00，period 为 10s，任务执行时长为 5s，那么 scheduleAtFixedRate 在 10:10 会执行下一次任务；如果任务执行时长为 15s，那么 scheduleAtFixedRate 在 10:15 会执行下一次任务。Timer 也是用固定速率执行周期性任务。scheduleWithFixedDelay 是固定延迟执行周期性任务，即相对于上一次任务结束时间往后推 period 时间后执行下一次任务。

使用 Timer 和 ScheduledExecutorService 实现类似在每周二 1:00:00 执行任务是比较麻烦的，此时，可以使用 Quartz 或者 Spring Task 实现。Spring Task 配置简单，在单实例任务场景下，笔者偏好于使用 Spring Task 实现。

```
<task:scheduler id="scheduler" pool-size="10"/>
<task:scheduled-tasks scheduler="scheduler">
```

```
    <!--每天两点执行-->
    <task:scheduled ref="relationClearTask" method="autoClearRelation"
cron="0 0 2 * * ?"/>
  </task:scheduled-tasks>
```

cron 配置表达式和 Quartz 基本一致，详细配置可以参考 Spring 官方文档。

14.7.2 分布式任务

使用上述机制进行单实例任务处理时是单点作业，如果实例失效了，那么任务可能得不到执行，另外，如果单实例任务处理遇到瓶颈，则不太容易做到动态扩容。因此，我们需要任务高可用和动态扩容，此时就需要分布式任务。使用分布式任务后，当一个实例失效，则可以将任务转移到其他实例进行处理。分布式任务支持任务分片，当任务处理遇到瓶颈，可以扩充任务实例来提升任务处理能力。

Quartz 支持任务的集群调度，如果一个实例失效，则可以漂移到其他实例进行处理，但是其不支持任务分片。tbschedule 和 elastic-job 除了支持集群调度特性，还支持任务分片，从而可以进行动态扩容/缩容。tbschedule 和 elastic-job 都能满足我们的场景需求。在本书出版时，唯品会也开源了基于 elastic-job 开发的 Saturn，京东内部也有相关实现，但在本书出版前暂未开源。本章选择 elastic-job 来讲解分布式任务。

14.7.3 Elastic-Job 简介

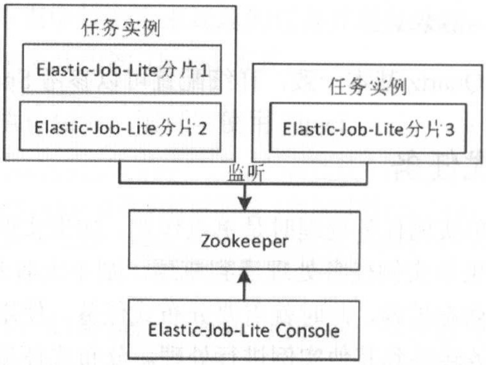
Elastic-Job 是当当开源的一款分布式任务调度框架，目前提供了两个独立子项目：Elastic-Job-Lite 和 Elastic-Job-Cloud。

Elastic-Job-Lite 定位为轻量级无中心解决方案，可以动态暂停/恢复任务实例，目前不支持动态扩容任务实例。Elastic-Job-Cloud 使用 Mesos + Docker 解决方案，可以根据任务负载来动态实现启动/停止任务实例，以及任务治理。Elastic-Job-Cloud 目前还处于开发中。Elastic-Job-Lite 和 Elastic-Job-Cloud 使用同一套任务 API，一次开发并根据需要以 Lite 或 Cloud 的方式部署。本书会重点介绍 Elastic-Job-Lite。关于 Elastic-Job-Cloud，可以从官网中了解其最新动向。

14.7.4 Elastic-Job-Lite 功能与架构

Elastic-Job-Lite 实现了分布式任务调度、动态扩容缩容、任务分片、失效转移、运维平台等功能。

1．整体架构



Elastic-Job-Lite 采用去中心化的调度方案，由 Elastic-Job-Lite 的客户端定时自动触发任务调度，通过任务分片的概念实现服务器负载的动态扩容/缩容，并且使用 ZooKeeper 作为分布式任务调度的注册和协调中心，当某任务实例崩溃后，自动失效转移，实现高可用，并提供了运维控制台，实现任务参数的动态修改。

2．任务分片

任务如果并行处理或者分布式处理，则需要使用任务分片，即把任务拆成 N 个子任务。比如，我们需要遍历某张数据库表，现在有 1 台服务器，为了实现多线程处理，此时可以将数据分片为 10 份，如 id % 10，那么会有 10 个线程并发处理这些任务，从而提升了处理性能。如果有两台服务器，并且还将数据分片为 10 份，如 id % 10，那么机器 1 会处理 1,3,5,7,9；机器 2 会处理 0,2,4,6,8；每台机器是 5 个线程并发处理任务。通过任务分片可以实现任务并发处理，通过增加机器可以实现动态扩容/缩容。

14.7.5 Elastic-Job-Lite 示例

1．启动 ZooKeeper

Elastic-Job 使用 ZooKeeper 进行分布式任务调度（本节使用的是 ZooKeeper-3.4.9），执行如下脚本后启动 ZooKeeper。

```
./bin/zkServer.sh start
```

2．添加 Elastic-Job-Lite 依赖

下面添加 Elastic-Job-Lite 依赖（本文使用最新的 Elastic-Job-Lite 2.x），其 API 与 1.x

完全不同。

```
<dependency>
   <groupId>com.dangdang</groupId>
   <artifactId>elastic-job-lite-core</artifactId>
   <version>2.0.0</version>
</dependency>
<dependency>
   <groupId>com.dangdang</groupId>
   <artifactId>elastic-job-lite-spring</artifactId>
   <version>2.0.0</version>
</dependency>
```

3. 任务开发

为了便于统一名词，在本书中不区分任务与作业。Elastic-Job 提供了三种类型的任务：Simple 类型任务（最简单的实现，支持分片特性）、Dataflow 类型任务（将任务的数据抓取和处理分离）和 Script 类型任务（脚本类型任务，如 Shell/Python 脚本等）。

Simple 类型任务

```
public class MySimpleJob implements SimpleJob {
   public void execute(ShardingContext shardingContext) {
      switch (shardingContext.getShardingItem()) {
         //任务按照主键 ID 分 3 片（ID % 3)
         case 0: //分片 0
            process(fetch(0, 3, 6, 9));
            break;
         case 1: //分片 1
            process(fetch(1, 4, 7));
            break;
         case 2: //分片 2
            process(fetch(2, 5, 8));
            break;
      }
   }
}
```

通过 Simple 类型任务实现 SimpleJob#execute 即可，然后再根据分片配置信息进行分片处理实现。

Dataflow 类型任务

```
public class MyDataflowJob implements DataflowJob<String> {
   public List<String> fetchData(ShardingContext shardingContext) {
```

```
        switch (shardingContext.getShardingItem()) {
            //任务按照主键 ID 分 3 片（ID % 3）
            case 0: //分片 0
                return fetch(0, 3, 6, 9);
            case 1: //分片 1
                return fetch(0, 3, 6, 9);
            case 2: //分片 2
                return fetch(0, 3, 6, 9);
        }
        return null;
    }
    public void processData(ShardingContext shardingContext, List<String> data) {
        //任务处理
    }
}
```

Dataflow 类型任务将任务分为抓取数据（fetchData）和处理数据（processData）两部分。其中，流式任务只有当 fetchData 方法返回值为 null 时，任务才停止抓取，否则任务将一直运行下去。非流式任务在每次任务执行过程中，只执行一次 fetchData 和 processData 方法。可以在任务配置时，设置是否是流式任务。

4. 任务配置与启动

Elastic-Job 支持 Java 配置和 Spring 配置文件配置任务，本文选择使用 Spring 配置文件配置。

配置 ZK 注册中心

```
<reg:ZooKeeper id="regCenter"
            server-lists="192.168.61.129:2181"
            namespace="my-job"
            connection-timeout-milliseconds="2000"
            session-timeout-milliseconds="3000"
            base-sleep-time-milliseconds="1000"
            max-sleep-time-milliseconds="3000"
            max-retries="3"/>
```

Elastic-Job 使用 Apache Curator 客户端来连接 ZK，配置参数如下所示。

server-lists：ZooKeeper 服务器列表，多个地址用逗号分隔。

namespace：当前注册中心使用的是 ZooKeeper 命名空间，不同类型的任务可以放到不同的命名空间。

connection-timeout-milliseconds：ZK 连接超时时间，默认为 15000ms。

session-timeout-milliseconds：ZK 会话超时时间，默认为 60000ms。

digest：连接 ZK 时的权限令牌，默认不需要权限验证。

base-sleep-time-milliseconds：使用 ExponentialBackoffRetry 指数退避算法重试时的初始重试时间，默认为 1000ms。

max-sleep-time-milliseconds：使用 ExponentialBackoffRetry 指数退避算法重试时的最大重试时间，默认为 3000ms。

max-retries：使用 ExponentialBackoffRetry 指数退避算法的最大重试次数。

配置 Simple 类型任务

```
<job:simple registry-center-ref="regCenter"
        id="mySimpleJob"
        class="com.elasticjob.MySimpleJob"
        cron="0/10 * * * * ?"
        sharding-total-count="3"
        sharding-item-parameters="0=A,1=B,2=C"
        job-parameter="pageSize=5"
        disabled="false"
        overwrite="false"/>
```

配置参数如下所示。

registry-center-ref：配置使用的注册中心。

id：任务/作业名称。

class：Simple 类型任务实现类。

cron：cron 表达式，配置作业触发时间，目前使用 Quartz 表达式。

sharding-total-count：总的任务分片数，通过它来实现任务并发执行和分布式。

sharding-item-parameters：分片序号和参数关系。

job-parameter：任务自定义参数，如每次查询数据库表的每页记录数，通过它可以实现动态分页参数配置。

diasbled：任务默认是否是禁止启动，当部署任务时需要先禁用，部署后统一启动时可以配置。

overwrite：是否使用本地参数配置覆盖注册中心配置，如果覆盖，那么任务启动时以本地配置参数为准。

job-sharding-strategy-class：任务分片算法，默认使用平均分配算法，可以进行算法的自定义。

这里要注意以下几点。

- 任务实例是以服务器ID+JOB ID作为唯一标识来区分的，即使一台服务器部署了多个实例，目前Elastic-Job也会把它们看作同一个实例，如果同一台服务器部署多个实例，则可能导致相同任务的重复执行。因此一台服务器应只部署一个任务实例。
- 任务的参数在任务实例第一次启动后注册到注册中心，之后任务实例重启后，任务参数将以注册中心的为准，更改本地配置是不起作用的，可以在任务控制台进行参数更改。如果配置了overwrite=true，则将以本地配置为准。
- sharding-item-parameters可以为不同的任务分片配置个性化参数，job-parameter可以为所有的任务分片配置参数。

任务类中的 ShardingContext 提供了任务分片上下文参数。

- **jobName：**任务名称/ID。
- **jobParameter：**任务自定义参数。
- **shardingTotalCount：**总的分片数量，可以根据它分别抓取任务数据。
- **shardingItem：**分配本任务实例的分片序号。
- **shardingParameter：**分配本任务实例的分片参数。

配置 Dataflow 类型任务

```
<job:dataflow registry-center-ref="regCenter"
            id="myDataflowJob"
            class="com.elasticjob.MyDataflowJob"
            cron="*/10 * * * * ?"
            streaming-process="false"
            sharding-total-count="3"
            sharding-item-parameters="0=A,1=B,2=C"
            job-parameter="pageSize=5"
            disabled="false"
            overwrite="false"/>
```

参数配置和 Simple 类型任务差不多，只是多了一个 streaming-process，然后判断其

配置是否是流式处理数据，如果是流式处理数据，那么 fetchData 方法返回空时，才结束执行任务。如果是非流式处理数据，那么只执行一次 fetchData 和 processData 方法，然后任务执行就结束了。

接着启动加载 Spring 配置文件的 JVM 实例，即可启动任务。

任务控制台

Elastic-Job-Lite 提供了 elastic-job-lite-console 控制台，用于动态配置任务。下载 elastic-job-lite-console 并部署到 Tomcat 中，然后启动即可。

添加注册中心

首先，需要添加注册中心，如下图所示。然后就可以对该注册中心的任务进行维护了。

任务配置

通过任务控制台可以进行任务/作业的参数动态更改，如下图所示。

至此使用 Elastic-Job-Lite 开发分布式任务就介绍完了。

15 队列术

队列，在数据结构中是一种线性表，从一端插入数据，然后从另一端删除数据。本书的目的不是讲解各种队列及如何实现，而是讲述在应用层面使用队列能解决哪些场景问题。

在我们的系统中，不是所有的处理都必须实时处理，不是所有的请求都必须实时反馈结果给用户，不是所有的请求都必须 100%一次性处理成功，不知道哪个系统依赖“我”来实现其业务处理，保证最终一致性，不需要强一致性。此时，我们应该考虑使用队列来解决这些问题。当然我们也要考虑是否需要保证消息处理的有序性及如何保证，是否能重复消费及如何保证重复消费的幂等性。在实际开发时，我们经常使用队列进行异步处理、系统解耦、数据同步、流量削峰、扩展性、缓冲等。

15.1 应用场景

异步处理：使用队列的一个主要原因是进行异步处理，比如，用户注册成功后，需要发送注册成功邮件/新用户积分/优惠券等；缓存过期时，先返回过期数据，然后异步更新缓存、异步写日志等。通过异步处理，可以提升主流程响应速度，而非主流程/非重要处理可以集中处理，这样还可以将任务聚合批量处理。因此，可以使用消息队列/任务队列来进行异步处理。

- **系统解耦**：比如，用户成功支付完成订单后，需要通知生产配货系统、发票系统、库存系统、推荐系统、搜索系统等进行业务处理，而未来需要支持哪些业务是不知道的，并且这些业务不需要实时处理、不需要强一致，只需要保证最终一致性即可，因此，可以通过消息队列/任务队列进行系统解耦。
- **数据同步**：比如，想把MySQL变更的数据同步到Redis，或者将MySQL的数据同步到Mongodb，或者让机房之间的数据同步，或者主从数据同步等，此时可以考虑使用databus、canal、otter等。使用数据总线队列进行数据同步的好处是可以保证数据修改的有序性。
- **流量削峰**：系统瓶颈一般在数据库上，比如扣减库存、下单等。此时可以考虑使用队列将变更请求暂时放入队列，通过缓存+队列暂存的方式将数据库流量削峰。同样，对于秒杀系统，下单服务会是该系统的瓶颈，此时，可以使用队列进行排队和限流，从而保护下单服务，通过队列暂存或者队列限流进行流量削峰。

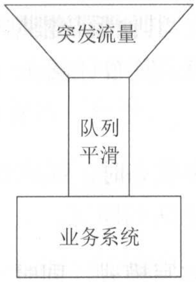
队列的应用场景非常多，以上只列举了一些常见用法和场景。

15.2 缓冲队列

典型的如 Log4j 日志缓冲区，当我们使用 log4j 记录日志时，可以配置字节缓冲区，字节缓存区满时，会立即同步到磁盘。Log4j 是使用 BufferedWriter 实现的。此模式不是异步写，在缓冲区满的时候还是会阻塞主线程。如果需要异步模式，则可以使用 AsyncAppender，然后通过 bufferSize 控制日志事件缓冲区大小。

同样，在电商进行大促时，此时的系统流量会高于平常流量的几倍甚至几十倍，此时应进行一些特殊的设计来保证系统平稳度过这段时期。而解决的手段很多，一般牺牲业务的强一致性，保证最终一致性即可。

如下图所示，使用缓冲队列应对突发流量时，并不能使处理速度变快，而是使处理速度变平滑，从而不会因瞬间压力太大而压垮应用。



通过缓冲区队列可以实现批量处理、异步处理和平滑流量。

15.3 任务队列

使用任务队列可以将一些不需要与主线程同步执行的任务扔到任务队列进行异步处理。笔者用得最多的是线程池任务队列（默认为 LinkedBlockingQueue）和 Disruptor 任务队列（RingBuffer）。如用户注册完成后，将发送邮件/送积分/送优惠券任务扔到任务队列进行异步处理；刷数据时，将任务扔到队列异步处理，处理成功后再异步通知用户。还有删除 SKU 操作，在用户请求时直接将任务分解并扔到队列进行异步处理，处理成功后异步通知用户。以及查询聚合时，将多个可并行处理的任务扔到队列，然后等待最慢的一个任务返回。

通过任务队列可以实现异步处理、任务分解/聚合处理。

注：JDK7 提供了 ExecutorService 的新实现 ForkJoinPool，其提供的 Work-stealing 机制，可以更好地提升并发效率。

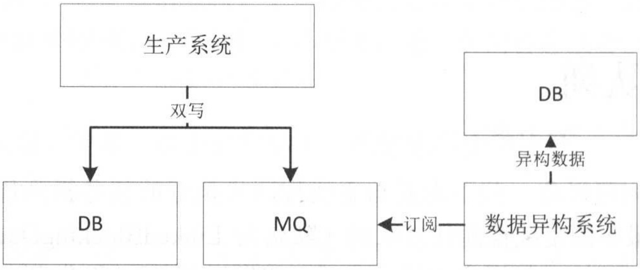
15.4 消息队列

笔者所在公司使用的系统是自主研发的 JMQ；开源的系统有 ActiveMQ、Kafka、

Redis。使用消息队列存储各业务数据，其他系统根据需要订阅即可。常见的订阅模式是：点对点（一个消息只有一个消费者）、发布订阅（一个消息可以有多个消费者）。而常用的是发布订阅模式。

比如，修改商品数据、变更订单状态时，都应该将变更信息发送到消息队列，如果其他系统有需要，则直接订阅该消息队列即可。

一般我们会在应用系统中采用双写模式，同时写 DB 和 MQ，然后异构系统可以订阅 MQ 进行业务处理（见下图）。因为在双写模式下没有事务保证，所以会出现数据不一致的情况，如果对一致性要求没那么严格，则这种模式是没问题的，而且在实际应用中这种模式也非常多。

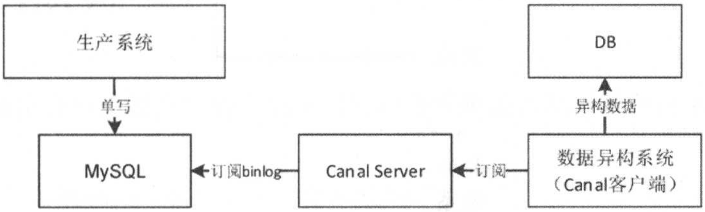


如下代码是双写示例，事务成功后发 MQ。

```
public OrderDTO create(final OrderDTO order) throws OrderException {
    OrderDTO createdOrderDTO = executeInShardingTrans((status) -> {
        //插入订单到 DB
        OrderDTO insertOrderDTO = convert(orderService.insert(order));
        return insertOrderDTO;
    }, order);
    //发 MQ
    orderMqProducer.publish(OrderMqType.CREATED, null, insertOrderDTO);
    //写缓存
    orderCache.put(createdOrderDTO);
    return createdOrderDTO;
}
```

如果在事务中发 MQ，会存在事务回滚，但是 MQ 发送成功了，则需要消息消费者进行幂等处理。如果事务提交慢，但是 MQ 已经发出去了，则此时根据 MQ 信息再去获取数据库数据可能不是最新的。如果 MQ 发送慢，则会导致事务无法快速提交，造成数据库堵塞。同样不要在事务中掺杂 RPC 调用，RPC 服务不稳定，同样会引起数据库阻塞。

也可以采用订阅数据库日志机制来实现数据库变更捕获，这样生产系统只需要单写DB，然后通过如 Canal 订阅数据库 binlog 实现数据库数据变更捕获，然后业务端订阅 Canal 进行业务处理。这种方式可以保证一致性。

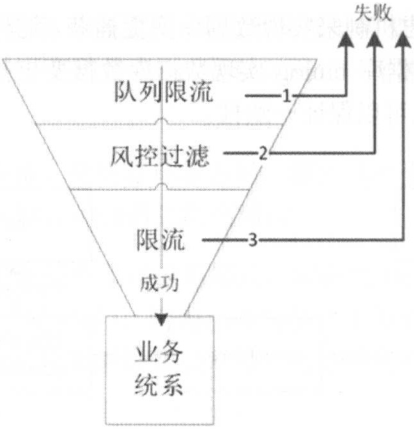


通过消息队列可以实现异步处理、系统解耦和数据异构。

15.5 请求队列

请求队列是指类似在 Web 环境下对用户请求排队，从而进行一些特殊控制：流量控制、请求分级、请求隔离。例如将请求按照功能划分到不同的队列，从而使得不同的队列出现问题后相互不影响。还可以对请求分级，一些重要的请求可以优先处理（发展到一定程度应将功能物理分离）。另外，服务器处理能力有限，在接近服务器瓶颈时需要考虑限流，最简单的限流是丢弃处理不了的请求，此时可以使用队列进行流量控制。

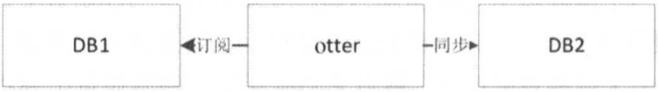
如下图所示，这里使用请求队列来实现漏斗模式，对请求进行排队、过滤、限流，经过这些步骤后，流入业务系统的流量就非常小了，这样业务系统就不会被突发的大量请求搞垮。队列限流可以通过队列大小（如果队列满了，就抛弃新的请求）和排队超时（队列里的请求很长时间没被处理）实现，如果失败了，则返回让客户重新排队或者重试。使用这种机制可以很好地保护系统不会受到突发流量的冲击。这种机制一般用于前端入口。



15.6 数据总线队列

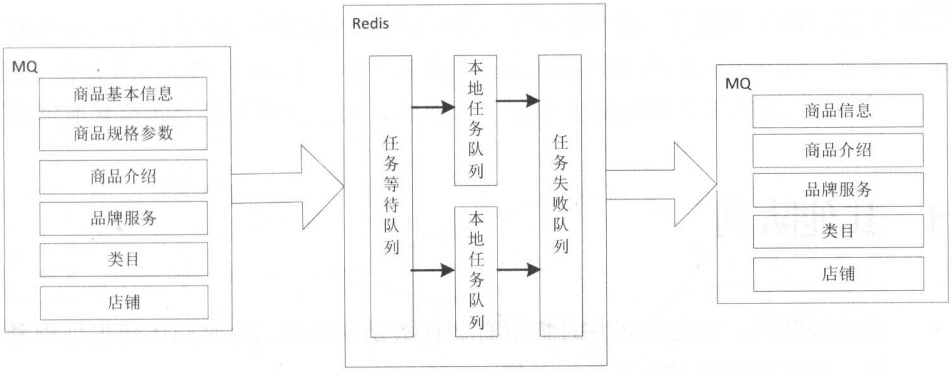
一般消息队列中的消息都是业务维度的简单数据，如业务键或业务状态。在商品信息变更场景中，当 SKU 信息变更了，只下发一个 SKU ID，订阅者需要再查一遍商品系统来获取最新的变更数据，进行如商品信息缓存同步。所以使用现有的消息队列方式很难只进行变更部分的推送并保证数据的有序性。而此种场景比较适合使用数据总线队列实现。例如数据库变更后需要同步数据到缓存，或者需要将一个机房的数据同步到另一个机房，只是数据维度的同步，此时应该使用数据总线队列，如阿里的 Canal、LinkedIn 的 databus。使用数据总线队列的好处是，可以保证数据的有序性。阿里的 otter 是基于 Canal 的一款分布式数据库同步系统，如果想实时进行多机房、多数据库数据增量同步，则可以使用 otter。如果需要全量离线数据同步，则可以使用 kettle。

可以通过 otter 订阅某个 DB 的某些表，然后同步到另一个数据库中。如果系统中存在一些基础数据，则可以使用这种方式进行同步（见下图）。



15.7 混合队列

在第 16 章中介绍过混合队列。如下图所示是使用混合队列来解决实际问题。



此处 MQ 是使用京东自主研发的 JMQ，消息是可靠持久化存储的。应用会按照不同的维度发布消息到 JMQ。下游应用接收到该消息后会将其放入 Redis 中，使用 Redis List 来存储这些任务。应用将 Redis 消息消费处理后，会按照不同的维度聚合商品消息，然后再次发送出去。

使用 Redis 队列的主要原因是想提升消息堆积能力和并发处理能力。另外，在使用 Redis 构建消息队列时，需要考虑因网络抖动造成的消息丢失问题，因为 Redis 是没有事务回滚的，或者说是没有确认机制的。我们使用如下方式防止消息丢失。

```
try {
    id = queueRedis.opsForList()
                .rightPopAndLeftPush(queueName, processingQueueName);
} catch (Exception e) {
    //发生了网络异常，需要把 processing 中的 id 再放回到 waiting queue 中
    String msg =
          queueName + " to " + processingQueueName + " rpoplpush error";
    LOG.error(msg, e);
    //报警代码
}
```

而对于失败我们会进行三次重试，重试失败后放入失败队列，而失败队列是具有防重功能的（从本地队列和失败队列排重），这里使用 Redis Lua 脚本实现。

```
static EventQueueScript ADD_TO_FAIL_QUEUE_REDIS_SCRIPT =
     new EventQueueScript(
       "redis.call('lrem',  KEYS[1],  1,  ARGV[1])  redis.call('lrem',
KEYS[2], 1, ARGV[1]) return redis.call('lpush', KEYS[2], ARGV[1])"
   );
```

Redis 的作者 Antirez 开发的内存分布式消息队列 Disque，是未来更好的内存消息队列选择。

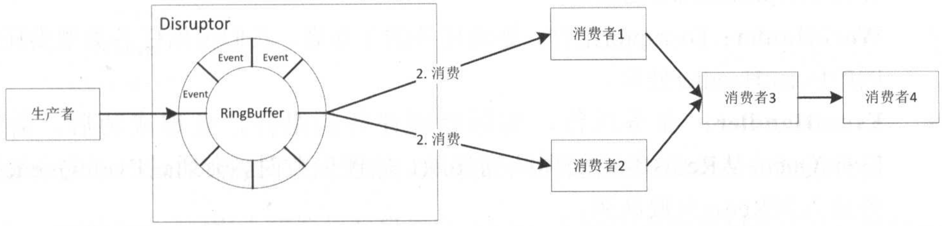
15.8 其他队列

- **优先级队列**：在实际开发时肯定有些任务是紧急的，此时应该优先处理紧急任务。所以请考虑对队列进行分级。
- **副本队列**：在进行一些系统重构或者上新的功能时，如果没有足够的信心保证业务逻辑正确，则可以考虑存储一份队列的副本（比如1小时、1天的消息），从而当业务出现问题时，可以对这些消息进行回放。
- **镜像队列**：每个队列不可能无限制被订阅消费，会有一个订阅量极限。当达到极限时，请考虑使用镜像队列方式解决该问题。
- **队列并发数**：不同队列实现，队列服务器端并发连接数是不一样的。一定不是增大队列并发连接数消费能力也随着增加，也不会因为增加了消费服务器消费，并发能力也随之增加，需要根据实际情况来设置合理的并发连接数。
- **推送拉取**：消息体内容不是越全越好，需要根据具体业务设计消息体。如有些系统依赖商品变更消息（只有一个SKU），有些系统依赖商品状态消息（SKU、状态），有些系统依赖商品属性变更消息（SKU、变更的属性）等。如果让所有系统都消费商品变更消息，那么这些系统都会调用商品查询服务，拉取最新商品信息，然后进行处理。因此，要根据实际情况来决定是使用推送方式（将系统需要的所有信息推过去）还是使用拉取方式（只推送ID，然后再查一遍）。

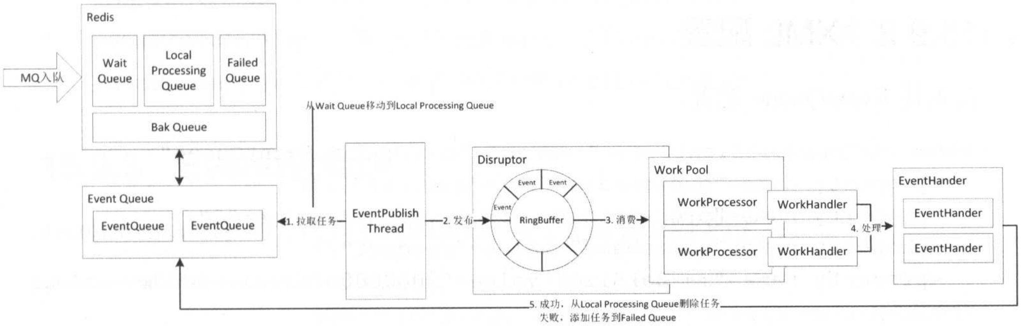
15.9 Disruptor+Redis 队列

15.9.1 简介

Disruptor 是 LMAX 开源的一个高性能异步处理框架，它提供了高性能无锁内存队列实现，并优化了 CPU 伪共享，用于构建低延迟高吞吐量的交易型应用。使用 Disruptor 也可以构建复杂的任务工作流，如下图所示，这里实现消费者工作流。



在实际项目中，我们使用 Disruptor 配合 Redis 来异步处理任务，整体架构如下图所示。



- **Redis队列**：我们使用Redis List数据结构来存储任务，并分为等待队列、本地处理队列、失败队列、备份队列。任务会首先发布到等待队列，然后会转移到本地处理队列进行处理，处理成功后会被从本地处理队列中移除，如果处理失败了，则会被移到失败队列等待人工介入。备份队列也叫作镜像队列，当遇到问题时，可以进行任务回放。我们使用Redis最高时有2亿多个任务在等待队列中等待处理。Redis队列中的任务可以是其他系统推送的，或者是MQ推送的。

- **EventQueue**：业务组件，封装了Redis队列的访问。
- **EventPublishThread**：业务组件，通过EventQueue拉取Redis Wait Queue任务，首先被移动到Local Processing Queue，然后被放入Disruptor RingBuffer内存队列处理。
- **RingBuffer**：Disruptor组件，一个环形队列，使用定长数组存储，并预先填充好任务/事件，不需要像链表那样每次添加/删除节点时去创建/回收节点，从而避免一定的垃圾回收。环形队列数组长度是2^*N*，可以使用位运算提升性能。整个队列使用无锁设计从而减少了竞争。通过缓存行填充解决CPU伪共享问题。
- **WorkPool**：Disruptor组件，存储WorkProcessor的池子，Disruptor将任务处理器放入WorkPool中，然后通过Executor并发启动每一个WorkProcessor。
- **WorkProceesor**：Disruptor组件，WorkProcessor从RingBuffer消费事件/任务，并交由WorkHandler处理。
- **WorkHanler**：Disruptor组件，处理任务的工作者，我们根据任务类型委托给不同的EventHandler处理。
- **EventHandler**：业务组件，实际处理任务的组件，处理成功后，会通过EventQueue从Redis本地处理队列移除。处理失败时，会通过EventQueue把任务放入到Redis失败队列。

接下来我们看一下这些组件的核心实现（本书使用的是 Disruptor 3.2.1）。

15.9.2 XML 配置

首先是 EventQueue 配置。

```
<bean id="productEventQueue" parent="com.queue.EventQueue">
   <property name="queueRedis" ref="queueRedis"/>
   <property name="processingErrorRetryCount" value="2"/>
   <property name="queueName" value="product"/>
   <property name="maxBakSize" value="20000000"/><!-- 2000w -->
</bean>
```

- **queueRedis**：配置Redis队列实例，我们使用jedis客户端。
- **processingErrorRetryCount**：本地任务队列中失败后的重试次数，当Disruptor处理失败时会进行重试。
- **queueName**：我们使用的Redis等待队列名称，任务会从该队列拉取，队列使用List数据结构实现。

- **maxBakSize**：队列的镜像大小，当从等待队列中拉取任务时，会放入一份镜像/备份队列，从而当业务处理出现问题时进行重放，业务实现时要考虑幂等性。

接下来是 EventHandler 配置。

```
<bean id="productEventHandler" class="com.task.handler. ProductEventHandler"/>
```

最后是 EventWorker 实现。

```
<bean id="productEventWorker" class="com.task.EventWorker"
                            init-method= "init" destroy-method="stop">
   <property name="threadPoolSize" value="256"/>
   <property name="ringBufferSize" value="4096"/>
   <property name="eventHandlerMap">
      <map>
        <entry key-ref="productEventQueue" value-ref=" productEventHandler"/>
        ......
      </map>
   </property>
</bean>
```

业务组件 EventWorker：用于创建 Disruptor 相关组件，包含如下配置项目。

- **init/stop**：init用于初始化并启动Disruptor。Stop用于当JVM终止时停止Disruptor组件。
- **ringBufferSize**：环形队列大小，大小必须是2的倍数。
- **eventHandlerMap**：映射 EventQueue 与 EventHandler 的关系，从特定的EventQueue中获取的任务将被关联的EventHandler处理。

15.9.3 EventWorker

eventHandlerMap 用于配置 EventQueue 与 EventHandler 的关系。

```
public void setEventHandlerMap(
                    Map<EventQueue, EventHander> eventHandlerMap) {
   this.eventHandlerMap = eventHandlerMap;
   if(MapUtils.isNotEmpty(eventHandlerMap)) {
      this.eventQueueMap = Maps.newHashMap();
      for(Map.Entry<EventQueue, EventHander> entry :
                                    eventHandlerMap.entrySet()) {
         EventQueue queue = entry.getKey();
         this.eventQueueMap.put(queue.getQueueName(), queue);
      }
   }
```

```
}
```

eventHandlerMap 存储了 EventQueue 和 EventHandler 的关系。eventQueueMap 存储了 queueName 与 EventQueue 的关系。

init

初始化 Disruptor、WorkHandler、EventHandler、EventPublishThread 等组件，并启动 Disruptor、EventPublishThread。

```
public void init() throws Exception {
    //1. 创建 Disruptor
    disruptor = new Disruptor<Event>(
            new DefaultEventFactory(), //使用默认事件工厂
            ringBufferSize,  //RingBuffer 大小
            Executors.newFixedThreadPool(threadPoolSize), //消费者线程池
            ProducerType.MULTI,  //支持多事件发布者
            new BlockingWaitStrategy()); //阻塞等待策略
    //2. 获取 RingBuffer
    ringBuffer = disruptor.getRingBuffer();
    //3. 处理异常
    disruptor.handleExceptionsWith(new ExceptionHandler() {
        ……
    });


    //4. 创建工作者处理器
    WorkHandler<Event> workHandler = new WorkHandler<Event>() {
        @Override
        public void onEvent(Event event) throws Exception {
            String type = event.getEventType();
            //4.1 根据事件类型获取 EventQueue
            EventQueue queue = eventQueueMap.get(type);
            //4.2 根据 EventQueue 获取该队列的事件处理器（XML 中配置了关心）
            EventHander hander = eventHandlerMap.get(queue);
            //4.3 交由 EventHandler 处理该事件
            hander.onEvent(event.getKey(), type, queue);
        }
    };
    //5.1 创建工作者处理器（数量为线程池大小）
    WorkHandler[] workerHanders = new WorkHandler[threadPoolSize];
    for (int i = 0; i < threadPoolSize; i++) {
        workerHanders[i] = workHandler;
    }
    //5.2 告知 Disruptor 由工作者处理器处理
```

```
    disruptor.handleEventsWithWorkerPool(workerHanders);
    //6. 启动 Disruptor
    disruptor.start();
    //7. 启动发布者线程（每个 EventQueue 一个，可以优化为只有一个）
   for (Map.Entry<String, EventQueue> eventQueueEntry :
                                       eventQueueMap.entrySet()) {
       String eventType = eventQueueEntry.getKey();
       EventQueue eventQueue = eventQueueEntry.getValue();
       //每个类型的队列创建一个发布者
       EventPublishThread thread =
           new EventPublishThread(eventType, eventQueue, ringBuffer);
       eventPublishThreads.add(thread);
       thread.start();
    }
}
```

此处我们配置的 Disruptor 支持多发布者，当 RingBuffer 满时使用阻塞等待策略。WorkHandler 会将 Event 交给相应的 EventHandler 处理。

当 JVM 停止时，需要停止 Disruptor 和 EventPublishThread。

```
public void stop() {
    //1. 停止发布者线程
    for(EventPublishThread thread : eventPublishThreads) {
        thread.shutdown();
    }
    //2. 停止 Disruptor
    disruptor.shutdown();
}
```

15.9.4 EventPublishThread

```
public void run() {
    while(running) {//当调用 shutdown 方法时，设置为 false 即可
        String nextKey = null;
        try {
            if(nextKey == null) {//从 EventQueue 获取下一个任务
                nextKey = eventQueue.next();
            }
            if(nextKey != null) {//发布到 RingBuffer
                ringBuffer.publishEvent(
                            EVENT_TRANSLATOR, nextKey, eventType);
            }
        } catch (Exception e) {
            logError(nextKey, e);
```

```
        }
    }
}
```

EventPublishThread 实现比较简单，eventQueue#next 方法将从 Redis 等待队列 POP 一个任务，然后推送到本地任务队列（该队列名称是：queueName + JVM 实例所在机器 IP），并发布到 Disruptor RingBuffer。

EVENT_TRANSLATOR 用于将参数转化为 Disruptor 的 Event 对象。

```
public void translateTo(Event event, long sequence, String key, String eventType) {
    event.setKey(key);
    event.setEventType(eventType);
}
```

15.9.5 EventHandler

以我们的 ProductEventHandler 为例，key 就是有变更的 skuId，我们根据 skuId 拉取最新的变更内容，然后更新到线上异构集群，如果成功，则从本地任务队列删除任务。如果失败，则将重试或者推送到失败队列。

```
public void onEvent(String skuId, String eventType, EventQueue queue) {
    try {
        //将 skuId 最新的变更内容更新到线上异构数据集群
        queue.success(skuId);
    } catch (Exception e) {
        queue.fail(skuId);
    }
}
```

至此，涉及任务处理的内容就都介绍完了，使用 Disruptor 可以快速构建一套内存任务处理逻辑。接下来我们来看一下 EventQueue 的实现。

15.9.6 EventQueue

1. next 方法

```
public String next() throws Exception {
    while (true) {
        //1. 暂停 Queue 消费
        PauseUtils.pauseQueue(queueName);

        String id = null;
```

```
        try {
            //2. 从等待 Queue POP，然后 PUSH 到本地处理队列
            id = queueRedis.opsForList()
                    .rightPopAndLeftPush(queueName, processingQueueName);
        } catch (Exception e) {
            //3. 发生了网络异常后告警，然后人工或定期检测，
            //将本地任务队列长时间未消费的任务推送回等待队列
            continue;
        }

        //4. 返回获取的任务
        if (id != null) {
            awaitInMillis = DEFAULT_AWAIT_IN_MILLIS;
            return id;
        }
        lock.lock();
        try {
            //如果没有任务，则休息一下稍后处理，防止死循环耗死 CPU
            if (awaitInMillis < 1000) {
                awaitInMillis = awaitInMillis + awaitInMillis;
            }
            notEmpty.await(awaitInMillis, TimeUnit.MILLISECONDS);
        } catch (Exception e) {
            //ignore
        } finally {
            lock.unlock();
        }
    }
}
```

next 用于从 Redis 等待队列获取任务并推送到本地处理队列，然后返回此任务。放入本地处理队列使用了 rightPopAndLeftPush，目的是防止因为网络异常导致任务丢失（因为 Redis 本身是没有事务的），当发生网络异常时需要告警，然后人工介入处理，或者启动一个 Worker 定期检查队列内容是否长时间未消费，如果长时间未消费，则应该再转移回等待队列处理。如果队列中没有任务，则应该短暂休息一会儿，然后重试，不要造成死循环耗死 CPU。

2. success 方法

```
public void success(String id) {
    queueRedis.opsForList().remove(processingQueueName, 0, id);
}
```

当任务成功处理后，从本地任务队列移除该任务。

3. fail 方法

```
public void fail(final String id) {
    final int failedCount =
              failedCache.getUnchecked(id).incrementAndGet();
    if (failedCount < processingErrorRetryCount) {
    //如果小于重试次数，则直接添加到等待队列尾
        ADD_TO_BACK_REDIS_SCRIPT.exec(queueRedis, Lists.newArrayList
(processingQueueName, queueName), id);
    } else {//如果超过失败重试次数，则加入失败队列
        ADD_TO_FAIL_QUEUE_REDIS_SCRIPT.exec(queueRedis, Lists.newArrayList
(processingQueueName, failedQueueName), id);
    }
}
```

fail 方法根据失败重试次数决定是放入等待队列重试，还是超过了失败重试次数直接转移到失败队列，然后告警，人工介入处理。

4. enqueueToBack 方法

该方法用于接收其他系统推送的任务，比如接收 MQ 消息，然后入队到 Redis 等待队列。其核心实现是通过 Redis Lua 脚本将任务加入队列，在加入队列时需要同时镜像一份放入到备份队列。

```
ENQUEUE_TO_LEFT_REDIS_SCRIPT.exec(queueRedis, Lists.newArrayList (queueName,
id, makeBakQueueName(), maxBakSizeStr));
```

这里用 ENQUEUE_TO_LEFT_REDIS_SCRIPT 实现 Lua 脚本。

```
static EventQueueScript ENQUEUE_TO_LEFT_REDIS_SCRIPT = new EventQueueScript(
        " local remCount = 0 if redis.call('llen', KEYS[1]) < 10000 then
remCount = redis.call('lrem', KEYS[1], 1, KEYS[2]) end redis.call('lpush',
KEYS[1], KEYS[2]) " +
        " if tonumber(KEYS[4]) <=0 then return nil end " +
        " if remCount > 0 then return nil end " +
        " local len = redis.call('llen', KEYS[3]) " +
        " if len > tonumber(KEYS[4]) then redis.call('lpop', KEYS[3]) end
redis.call('rpush', KEYS[3], KEYS[2]) "
);
```

如果等待队列数量小于 10000，则会进行排重（通过 lrem 先删除，然后再通过 lpush 进行重排）。如果等待队列数量大于 10000，因为遍历 List 性能会变得很差，则此时不会进行排重。数据同时会被放入备份队列，当备份队列满了时，使用 FIFO 移除最先插入的任务。

5. 队列名称

- **queueName：** 即等待队列名称，在XML配置文件中配置了。
- **processingQueueName：** 本地处理队列名为queueName + "_processing_queue_" + localIp。
- **failedQueueName：** 失败队列名为queueName + "_failed_queue"。
- **bakQueueName：** 备份队列名为queueName + _"bak_queue_" + LocalTime.now ().getHour()，一个小时一个队列，这样一天就有24个备份队列。

至此，整个 Disruptor+Redis 实现的队列及任务处理逻辑就介绍完了，本章的示例实现并不完美，还有很多优化空间，尤其在排重、任务调度、自动化、可靠性等方面还可以优化得更好。

使用 Redis 会存在丢任务的风险，要根据实际业务来决定是否允许丢任务，实际上大多数业务只要保证尽量不丢即可，不需要保证百分之百不丢，实现百分之百不丢是非常难的。保证业务逻辑一定是正确的更重要，一旦业务逻辑写错了，就要想办法进行数据回滚，此时备份队列的数据就很有作用了，尤其是在重构或者有新的逻辑时。

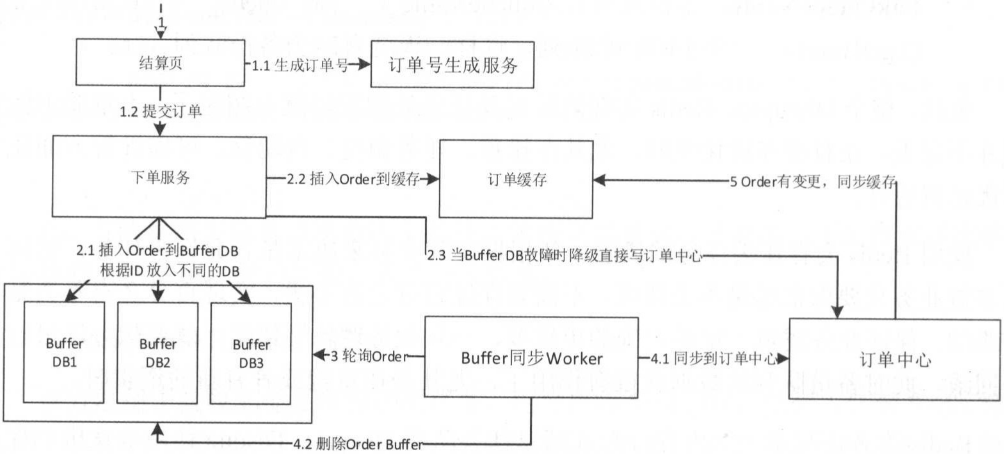
Redis 作者还写了一个内存分布式消息任务队列 Disque，Disque 使用确认机制保证消息可靠，目前只有 Beta 版本，不过截至目前已经快一年未更新了。

15.10 下单系统水平可扩展架构

订单系统是交易型网站的核心之一，用户会在这类网站上浏览并购买商品，购买后就会产生订单，接着需要用户进行支付，支付成功后就进入生产流程。而这其中最重要的一步就是能让用户先下单并成功支付，而后续流程可以不用实时处理。因此，如何保证下单功能的高性能和高可用是一个交易型网站的核心之一，当然这对于其他系统也同等重要。

一般订单系统会进行分库分表，如果分库分表的数量不够，则会影响到系统的性能，一般通过扩容来解决。或者当同一个订单库被多个系统依赖，其中某个系统有慢操作时，以及当一次下单需要写很多表并且订单量较大时，这都会造成用户下单速度变慢，甚至

无法下单。因此需要一种方案来解决这个问题。第 15 章介绍过缓冲队列，如果把订单放入缓冲队列，然后能迅速同步到订单中心，那么就可以把下单逻辑和操作订单逻辑分开，用户下单只操作缓冲表，而操作订单只操作订单表，从而在操作订单表时不会影响到缓冲表。而且缓冲表可以通过水平扩容来支持更大请求。下图是我们的订单系统的整体架构。



整体流程介绍如下。

1. 首先，用户在结算页提交订单后，系统调用订单号生成服务，然后结算服务会进行一些业务处理，最后调用下单服务提交订单。

2. 下单服务将订单写入订单缓冲表，下单服务和订单缓存表可以水平扩展，从而支持更多的下单操作。写入缓冲表成功后，将订单写入缓存，从而前端用户可以查看到当前订单。如果下单服务有问题，则可以考虑直接降级将订单写入订单中心。

3. 接着缓冲同步 Worker 轮询这些缓冲表。

4. 同步 Worker 将订单同步到订单中心，如果订单中心数据有变更，则更新订单缓存。

15.10.1 下单服务

```
public void submitOrder(OrderDTO order) {
    RoundRobinTable.Table table = roundRobinTable.nextTable();
    String sql = getSql(table);
    JdbcTemplate template = new JdbcTemplate(table.getDataSource());
    Long orderId = order.getId();
```

```
        String orderJson = JSONUtils.toJSON(order);
        template.update(sql, orderId, orderJson);
        //放入缓存
        orderCache.put(order);
    }
```

RoundRobinTable 是轮询选择下一个要写入的表，然后将数据写入到缓冲表，写入成功后再写入缓存。

此处的缓冲表结构可以包括：订单 ID、订单 JSON 串、订单状态、创建时间、处理状态、重试次数和 WorkerIP。

缓冲表所在的宿主机器也可能会出现硬件故障，当出现问题时，也只是影响部分订单的同步，不影响支付（因为缓存里有一份订单数据）。当性能遇到瓶颈时，可以通过水平扩容更多的缓冲表来解决。

15.10.2 同步 Worker

这里的同步 Worker 也用到了 Disruptor 架构，只是队列使用了数据库缓冲表来实现。

1. OrderBufferPublishThread

批量查询缓冲表并发布到 Disruptor RingBuffer 中。

```
Map<RoundRobinTable.Table, Long> lastIdMap = Maps.newHashMap();
Map<RoundRobinTable.Table, Object> lastOrderIdMap = Maps.newHashMap();
while(running) {
    RoundRobinTable.Table table = roundRobinTable.nextTable();
    //批量查询缓冲表（把处理状态改成“处理中”，并将 WorkerIp 设置为当前 JVM IP）
  List<Map<String, Object>> list =
                            listOrderBuffers(table, lastIdMap.get (table));
    //循环发布缓冲订单
    list.forEach((map) -> {
        Long id = (Long)map.get("id");
        Long orderId = (Long)map.get("order_id");
        String orderJson = (String)map.get("order_json");
        publishEvent(table, id, orderId, orderJson);
        lastIdMap.put(table, id);
        lastOrderIdMap.put(table, orderId);
        tryRateLimit();//是否限流
    });
}
```

2. OrderBufferHandler

缓冲订单处理器将缓冲订单同步到订单中心。

```
Long orderId = orderBufferEvent.getOrderId();
String orderJson = orderBufferEvent.getOrderJson();
RoundRobinTable.Table table = orderBufferEvent.getTable();
try {
    //1. 同步缓冲订单到订单中心
    OrderDTO order = JSONUtils.fromJson(orderJson, OrderDTO.class);
    orderJsfService.save(order);
    //2. 操作成功删除缓冲订单
    deleteOrderBuffer(table, orderId);
} catch (OrderException e) {
    //3. 如果遇到异常，则首先判断是否已经插入数据库，如果是，则直接删除缓冲订单即可
    OrderDTO order = orderJsfService.getOrderFromDB(orderId);
    if (order != null) {
    //已经成功插入数据库
        deleteOrderBuffer(table, orderId);
    }
}
```

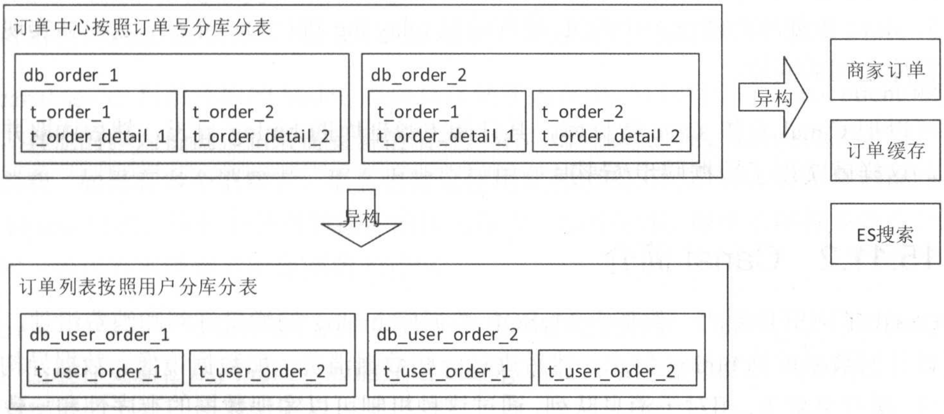
至此，一个简单的数据库订单缓冲架构就实现了，在实际生产环境中还需要进行健壮性设计，比如，Worker 多实例部署、Worker 不可用之后如何把它处理的订单快速恢复、缓存库不可用后的降级处理、订单号生成服务如何高可用等。

15.11 基于 Canal 实现数据异构

在大型网站架构中，DB 都会采用分库分表来解决容量和性能问题，但是分库分表之后带来了新的问题，比如不同维度的查询或者聚合查询，此时就会非常棘手。一般我们会通过数据异构机制来解决此问题。

如下图所示，为了提升系统的接单能力，我们会对订单表进行分库分表，但是，随之而来的问题是：用户怎么查询自己的订单列表呢？一种办法是扫描所有的订单表，然后进行聚合，但是这种方式在大流量系统架构中肯定是不行的。另一种办法是双写，但是双写的一致性又没法保证。还有一种办法就是订阅数据库变更日志，比如订阅 MySQL

的 binlog 日志模拟数据库的主从同步机制，然后解析变更日志将数据写到订单列表，从而实现数据异构，这种机制也能保证数据的一致性。

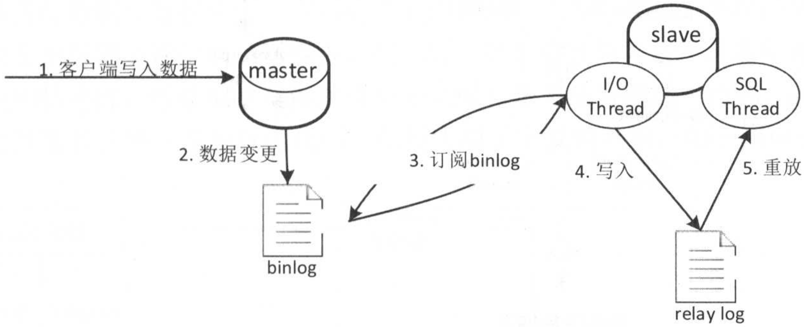


除了可以进行订单列表的异构，像商家维度的异构、ES 搜索异构、订单缓存异构等都可以通过这种方式解决。

在介绍 Canal 之前，我们先看一下 MySQL 的主从复制架构。

15.11.1 MySQL 主从复制

MySQL 主从复制架构如下图所示。



1．首先 MySQL 客户端将数据写入 master 数据库。

2．master 数据库会将变更的记录数据写入二进制日志中，即 binlog。

3．slave 数据库会订阅 master 数据库的 binlog 日志，通过一个 I/O 线程从 binlog 的指定位置拉取日志进行主从同步，此时 master 数据库会有一个 Binlog Dump 线程来读取

binlog 日志与 slave I/O 线程进行数据同步。

4．slave I/O 线程读取到日志后会先写入 relay log 重放日志中。

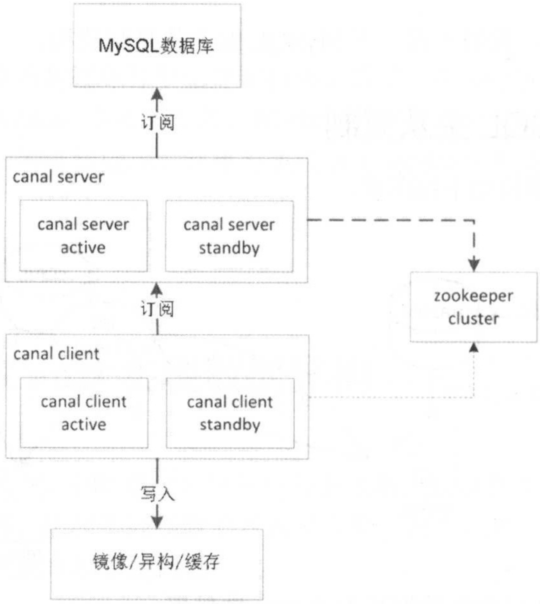
5．slave 数据库会通过一个 SQL 线程读取 relay log 进行日志重放，这样就实现了主从数据库之间的同步。

可以把 Canal 看作 slave 数据库，其订阅主数据库的 binlog 日志，然后读取并解析日志，这样就实现了数据同步/异构。

15.11.2 Canal 简介

Canal 是阿里开源的一款基于 MySQL 数据库 binlog 的增量订阅和消费组件，通过它可以订阅数据库的 binlog 日志，然后进行一些数据消费，如数据镜像、数据异构、数据索引、缓存更新等。相对于消息队列，通过这种机制可以实现数据的有序性和一致性。

Canal 架构如下图所示。

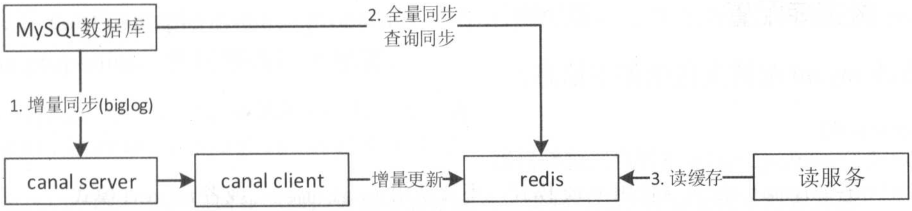


首先需要部署 canal server，可以同时部署多台，但是只有一台是活跃的，其他的作为备机。canal server 会通过 slave 机制订阅数据库的 binlog 日志。canal server 的高可用是通过来 zk 维护的。

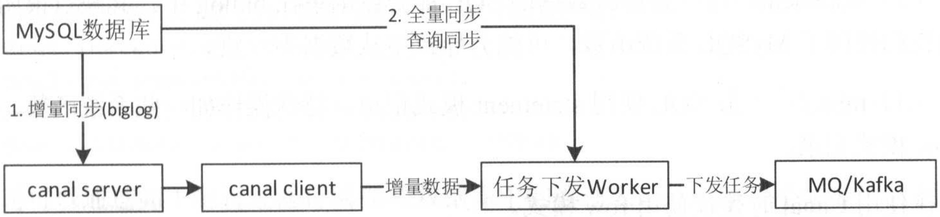
然后canal client会订阅canal server，消费变更的表数据，然后写入到如镜像数据库、异构数据库、缓存数据库，具体如何应用就看自己的场景了，同时也只有一台canal client是活跃的，其他的作为备机，当活跃的canal client不可用后，备机会被激活。canal client的高可用也是通过zk来维护的，比如zk维护了当前消费到的日志位置。

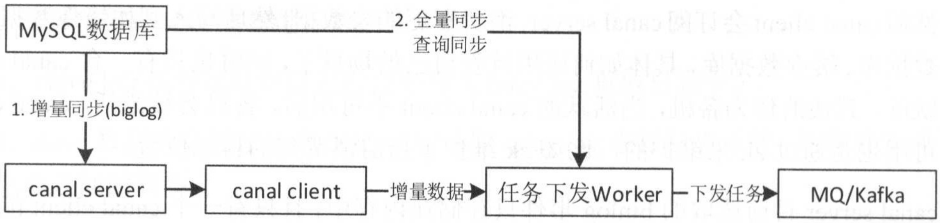
canal server目前读取的binlog事件只存储在内存中，且只有一个canal client能进行消费，其他的作为备机。如果需要多消费客户端，则可以先写入ActiveMQ/kafka，然后进行消费。如果有多个消费者，那么也建议使用此种模式，而不是启动多个canal server读取binlog日志，这样会使得数据库的压力较大。ActiveMQ提供了虚拟主题的概念，支持同一份内容多消费者镜像消费的特性。

canal 一个常见应用场景是同步缓存，当数据库变更后通过binlog进行缓存的增量更新。当缓存更新出现问题时，应能回退binlog到过去某个位置进行重新同步，并提供全量刷缓存的方法，如下图所示。



另一个常见应用场景是下发任务，当数据变更时需要通知其他依赖系统。其原理是任务系统监听数据库数据变更，然后将变更的数据写入MQ/Kafka进行任务下发，比如商品数据变更后需要通知商品详情页、列表页、搜索页等相关系统。这种方式可以保证数据下发的精确性，通过MQ发消息通知变更缓存是无法做到这一点的，而且业务系统中也不会散落着各种下发MQ的代码，从而实现了下发的归集，如下图所示。





类似于数据库触发器，只要想在数据库数据变更时进行一些处理，都可以使用 Canal 来完成。

在 MySQL 主从架构中，当有多个 slave 连接 master 数据库时，master 数据库的压力比较大，为保障 master 数据库的性能，canal server 可订阅 slave 的 binlog 日志，即是 slave 的 slave。

15.11.3 Canal 示例

1. 数据库配置

修改 my.ini 配置文件的如下信息。

```
[mysqld]
log-bin=mysql-bin #开启二进制日志
binlog-format=ROW #使用 row 模式，不要使用 statement 或者 mixed 模式
server_id=1 #配置主数据库 ID，不能和从数据库重复
```

binlog 提供了三种记录模式。

（1）row：记录的是修改的记录信息，而不是执行的 SQL，二进制日志文件会占用更大的空间，当执行 alter table 修改表结构造成记录变更时，该表的每一条记录都会被记录到日志中。

（2）statement：每一条修改数据的 SQL 都会被记录在 binlog 中，其缺点很明显，比如我们使用了 MySQL 系统函数，可能会导致主从数据不一致。

（3）mixed：一般 SQL 使用 statement 模式记录，特殊操作如一些系统函数，则采用 row 模式记录。

在使用 Canal 时建议使用 row 模式。

另外，在 MySQL 中执行“show binary logs”将看到当前有哪些二进制日志文件及其大小。

接下来，我们要为Canal创建一个复制账号，并为其授权查询和复制权限。

```
CREATE USER canal IDENTIFIED BY 'canal';
GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'canal'@'%';
```

这样我们就可以使用Canal这个数据库账号进行主从复制了。

2. 启动ZooKeeper

到zk官网下载ZooKeeper-3.4.9，如果需要修改zoo.cfg配置文件，则进行一些配置，然后执行如下命令启动单ZooKeeper服务器。

```
./bin/zkServer.sh  start
```

为了简化演示，我们没有部署zk集群。

3. Canal Server

到Canal官网下载**canal.deployer-1.0.22.tar.gz**，首先需要进行数据库实例的配置，其提供了conf/example/instance.properties一个示例配置，我们复制一份到conf/product/instance.properties，然后修改以下配置。

```
## mysql serverId 必须和 master 不一样
canal.instance.mysql.slaveId = 101

# position info  链接的数据库地址和从哪个二进制日志文件和从哪个位置开始
canal.instance.master.address = 192.168.0.10:3306
# MySQL 主库链接时，起始的 binlog 文件
canal.instance.master.journal.name =
# MySQL 主库链接时，起始的 binlog 偏移量
canal.instance.master.position =
# MySQL 主库链接时，起始的 binlog 的时间戳
canal.instance.master.timestamp =

# 用户名/密码/默认数据库/数据库编码（一定要配置正确）
canal.instance.dbUsername = canal
canal.instance.dbPassword = canal
canal.instance.defaultDatabaseName =
canal.instance.connectionCharset = UTF-8
```

还可以通过如下配置过滤订阅哪些数据库中的哪些表，从而减少不必要的订阅，比如，我们只关注产品数据库，那么通过如下模式即可只订阅产品数据库。

```
canal.instance.filter.regex = product_\d+\\.*
```

如果有多个数据库可以进行多个***/instance.properties 配置，则每个数据库设置一个配置文件。

接下来，进行 canal server 的配置，修改 conf/canal.properties。

```
#canal id、地址、端口和使用的 zk 服务地址
canal.id= 1
canal.ip=
canal.port= 11111
canal.zkServers=127.0.0.1:2181

#当前 canal server 上部署的实例，配置多个时用逗号分隔，此处配置了 product
canal.destinations= product

#使用 zk 持久化模式，这样可以保证集群数据共享，支持 HA
canal.instance.global.spring.xml = classpath:spring/ default-instance.xml
```

然后执行如下命令，启动一个 canal server。

```
./bin/startup.sh
```

4. Canal Client

接着创建或在已有的 Java 应用中添加 MySQL 客户端依赖、Canal 客户端依赖（com.alibaba.otter# canal.client# 1.0.22）。

订阅数据库变更的 Java 代码。

```
public void test() throws Exception {
    //通过 zookeeper 连接 canal server
    String zkServers = "192.168.61.129:2181";
    //目标是 product 实例
    String destination = "product";
   CanalConnector connector =
     CanalConnectors.newClusterConnector (zkServers, destination, "", "");

    //连接，并订阅 product 数据库下的 product 表（如果不写该模式，则订阅所有的）
    connector.connect();
    connector.subscribe("product_.*\\.product_.*");

    while (true) {
        //批量获取 1000 个日志（不确认模式）
        Message message = connector.getWithoutAck(1000);
        for(Entry entry : message.getEntries()) {
            //如果是行数据
```

```
            if(entry.getEntryType() == EntryType.ROWDATA) {
                //则解析行变更
                RowChange row = RowChange.parseFrom (entry.getStoreValue());
                EventType rowEventType = row.getEventType();
                for(RowData  rowData : row.getRowDatasList()) {
                    //如果是删除，则获取删除的数据，然后进行业务处理
                    if (rowEventType == EventType.DELETE) {
                        List<Column> columns = rowData. getBeforeColumnsList();
                        delete(columns);
                    }
                    //如果是新增/修改，则获取新增/修改的数据进行业务处理
                    if (rowEventType == EventType.INSERT
                            || rowEventType == EventType.UPDATE) {
                       List<Column> columns =
                                           rowData. getAfterColumnsList();
                        save(columns);
                    }
                }
            }
        }
        //确认日志消费成功
        connector.ack(message.getId());
    }
}
private static void save(List<Column> columns) {
    columns.forEach((column -> {
        String name = column.getName();
        String value = column.getValue();
        //业务处理
    }));
}
```

通过如上代码，我们就捕获了数据库日志变更，然后进行相关的业务处理即可。不管是数据异构还是缓存更新，因为数据就在这里，怎么处理就是业务逻辑的事情了。

京东内部有一个类似的组件 BinLake，截止本书出版前暂未开源。Canal 开源版本只提供了 MySQL 日志解析，如果想要 Oracle 日志解析，则可以使用 LinkedIn 的 Databus。

第4部分　案例

- 构建需求响应式亿级商品详情页
- 京东商品详情页服务闭环实践
- 使用 OpenResty 开发高性能 Web 应用
- 应用数据静态化架构高性能单页 Web 应用
- 使用 OpenResty 开发 Web 服务
- 使用 OpenResty 开发商品详情页

16 构建需求响应式亿级商品详情页

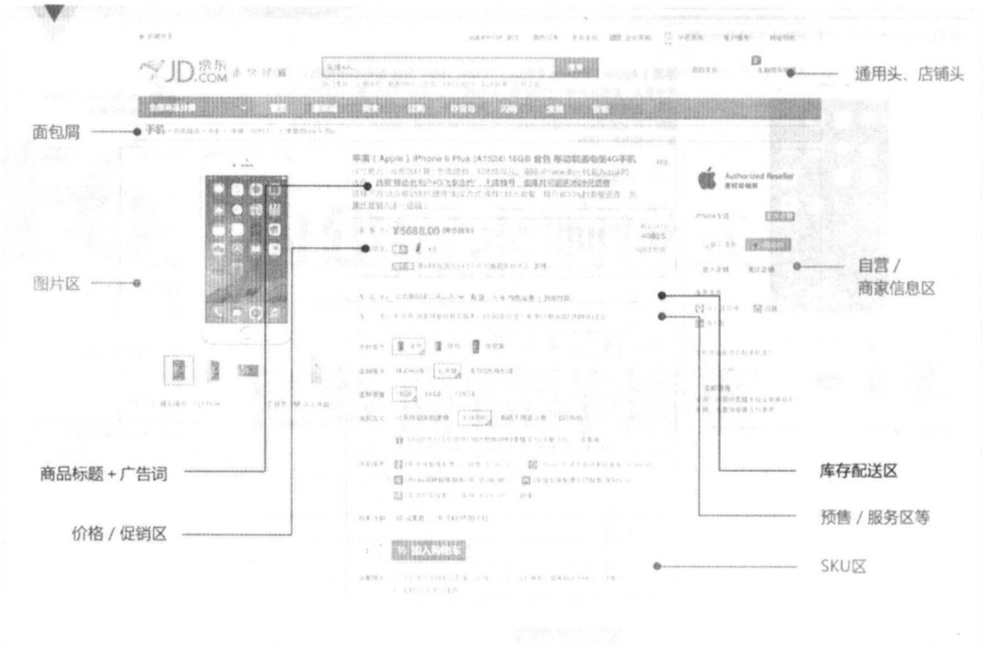
16.1 商品详情页是什么

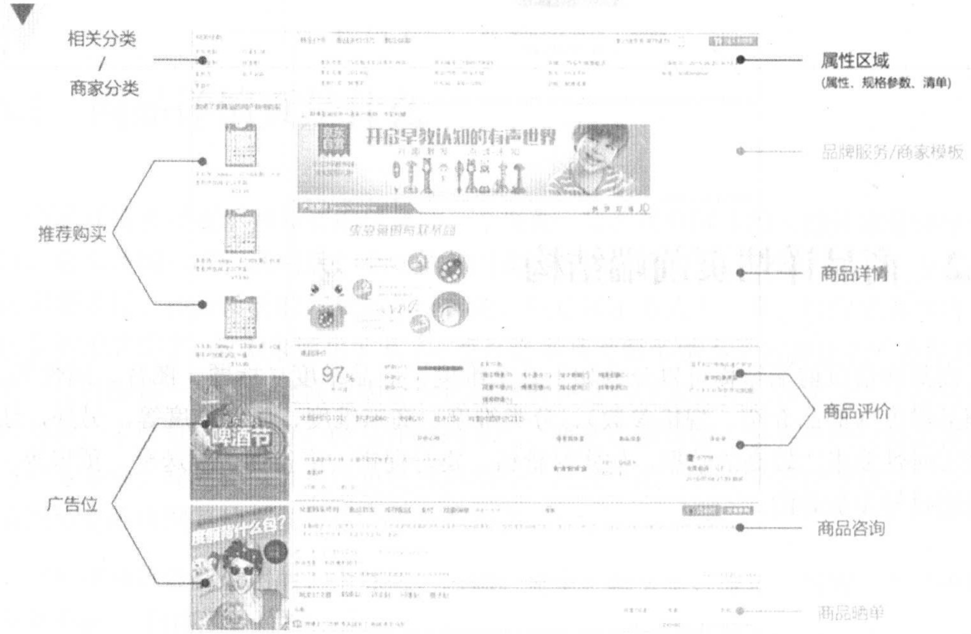
商品详情页是展示商品详细信息的一个页面，其承载着网站的大部分流量和订单的入口。京东商城目前有通用版、全球购、闪购、易车、惠买车、服装、拼购、今日抄底等许多套模板。各套模板的元数据是一样的，只是展示方式不一样。目前商品详情页的个性化需求非常多，数据来源也非常多，而且许多基础服务做不了的都放我们系统这里，因此，我们需要一种架构能快速响应和优雅地解决这些需求。我们重新设计了商品详情页的架构，主要包括三部分：商品详情页系统、商品详情页统一服务系统和商品详情页动态服务系统。商品详情页系统负责静的部分，而统一服务系统负责动的部分，动态服务系统负责给内网其他系统提供一些数据服务。

京东商城目前有通用版、全球购、闪购、易车、惠买车、服装、拼购、今日抄底等许多套模板。通用版如下图所示。

16.2 商品详情页前端结构

商品详情页前端结构可以分为如下几个维度：商品维度（标题、图片、属性等）、主商品维度（商品介绍、规格参数）、分类维度、商家维度、店铺维度等。另外，还有一些实时性要求比较高的数据，如实时价格、实时促销、广告词、配送至、预售等，它们是通过异步加载的。

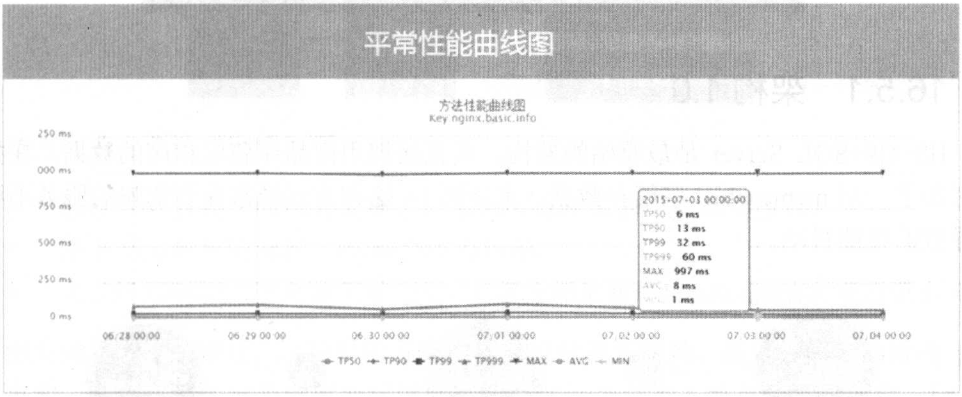




京东商城还有一些特殊维度数据，比如套装、手机合约机等数据，这些数据是主商品数据外挂的。

16.3 我们的性能数据

在“6·18”当天，京东商城的 PV 达到数亿次，当天服务器端的响应时间小于 38ms。如下图所示，这里展示的是第 1000 次采样数据排序后的第 99 次的时间。

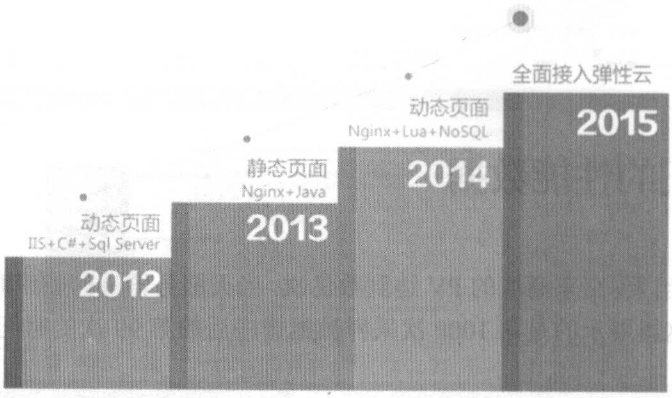


16.4 单品页流量特点

单品页流量的特点是离散数据、热点少，可以使用各种爬虫、比价软件抓取。

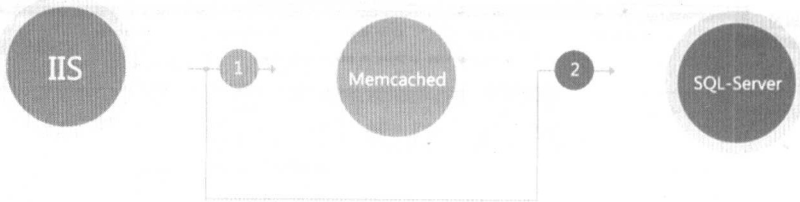
16.5 单品页技术架构发展

单品页技术架构的发展过程如下图所示。



16.5.1 架构 1.0

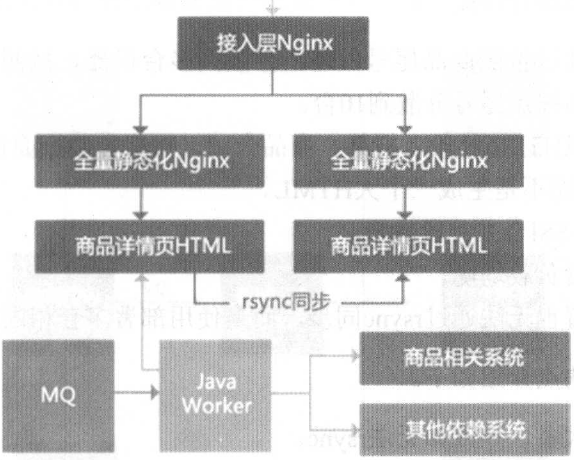
IIS+C#+SQL Server 是最原始的架构，其直接调用商品库获取相应的数据，在扛不住时加了一层 memcached 来缓存数据（见下图）。这种方式经常受到依赖的服务不稳定而导致的性能抖动。



16.5.2 架构 2.0

如下图所示的架构方案使用了静态化技术，按照商品维度生成静态化 HTML。该方案的主要思路介绍如下。

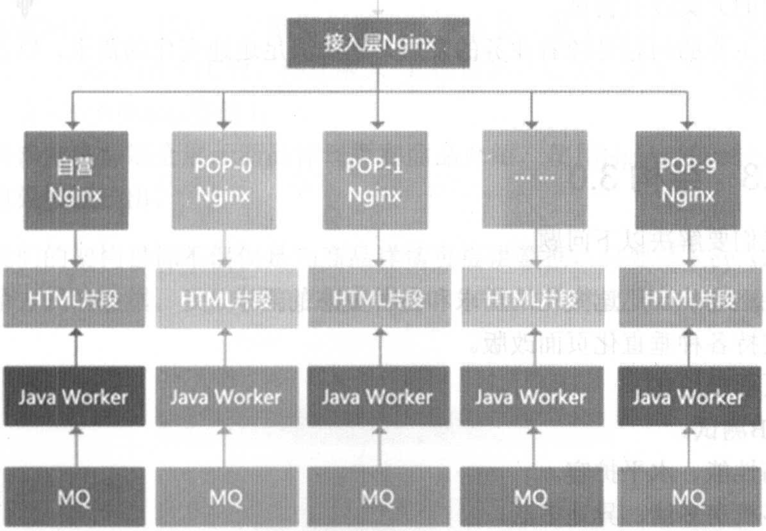
- 通过MQ得到变更通知。
- 通过Java Worker调用多个依赖系统生成详情页HTML。
- 通过rsync同步到其他机器。
- 通过Nginx直接输出静态页。
- 接入层负责负载均衡。



该方案的主要缺点介绍如下。

- 假设只有分类、面包屑变更了，那么所有相关的商品都要重刷。
- 随着商品数量的增加，rsync会成为瓶颈。
- 无法迅速响应一些页面需求变更，大部分都是通过JavaScript动态更改页面元素。

随着商品数量的增加，这种架构方案的存储容量遇到瓶颈，而且按照商品维度生成整个页面，会存在例如分类维度变更就要刷新一遍这个分类下所有信息的问题，因此，我们又改造了此架构方案，按照尾号路由到多台机器（见下图）。



此方案的主要思路介绍如下。

- 容量问题通过按照商品尾号做路由分散到多台机器，按照自营商品单独一台，第三方商品按照尾号分散到10台。
- 按维度生成HTML片段（框架、商品介绍、规格参数、面包屑、相关分类、店铺信息），而不是生成一个大HTML。
- 通过Nginx SSI合并片段输出。
- 接入层负责负载均衡。
- 多机房部署也无法通过rsync同步，而是使用部署多套相同的架构来实现。

该方案的主要缺点介绍如下。

- 碎片文件太多，导致如无法rsync。
- HDD做SSI合并时，高并发性能差，此时我们还没有尝试使用SSD。
- 模板如果要变更，则数亿件商品需要数天才能刷新完。
- 到达容量瓶颈时，我们会删除一部分静态化商品，然后通过动态渲染输出。动态渲染系统在流量高峰时会导致依赖系统压力大，抗不住。
- 还是无法迅速响应一些业务需求。

我们的痛点包括以下两点。

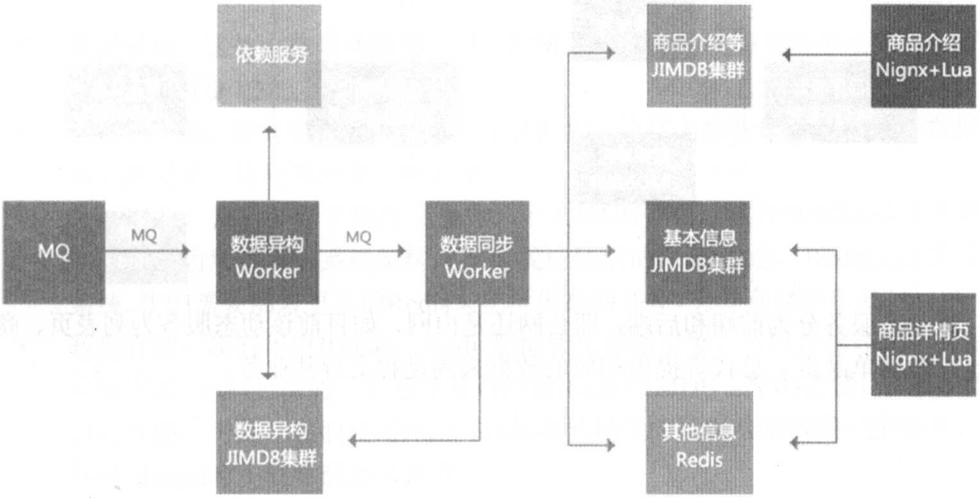
- 之前架构的问题存在容量问题，很快就会出现无法全量静态化，还是需要动态渲染；不过，对于全量静态化，可以通过分布式文件系统解决该问题，这里介绍的方案没有尝试。
- 最主要的问题是随着业务的发展，无法满足迅速变化的需求，以及一些变态的需求。

16.5.3 架构 3.0

现在我们要解决以下问题。

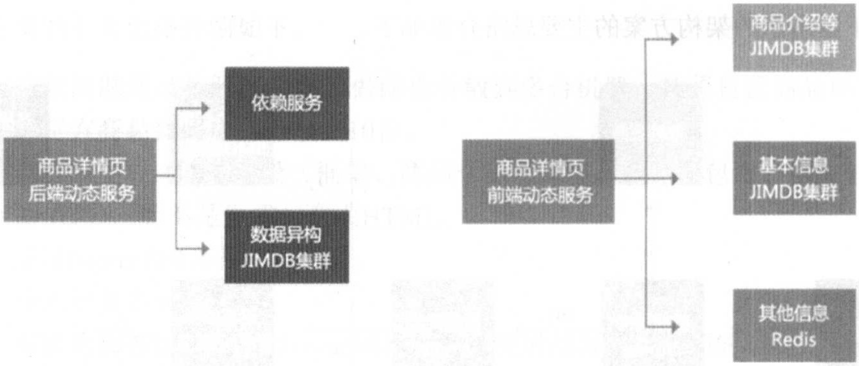
- 能迅速响应迅速变化的需求和各种变态的需求。
- 支持各种垂直化页面改版。
- 页面模块化。
- AB测试。
- 高性能、水平扩容。
- 多机房多活、异地多活。

如下图所示的架构方案的主要思路介绍如下。



- 数据变更还是通过MQ通知。
- 数据异构Worker得到通知，然后按照一些维度进行数据存储，存储到数据异构JIMDB集群（JIMDB：Redis+持久化引擎）中，存储的数据都是未加工的原子化数据，如商品基本信息、商品扩展属性、商品其他的一些相关信息、商品规格参数、分类、商家信息等。
- 数据异构Worker存储成功后，会发送一个MQ给数据同步Worker，数据同步Worker也可以叫作数据聚合Worker，其按照相应的维度聚合数据并存储到相应的JIMDB集群。其中三个维度包括：基本信息（基本信息+扩展属性等的一个聚合）、商品介绍（PC版、移动版）、其他信息（分类、商家等维度，其数据量小，直接使用Redis存储）。
- 此架构方案前端会展示商品详情页和商品介绍，使用Nginx+Lua技术获取数据并渲染模板输出。

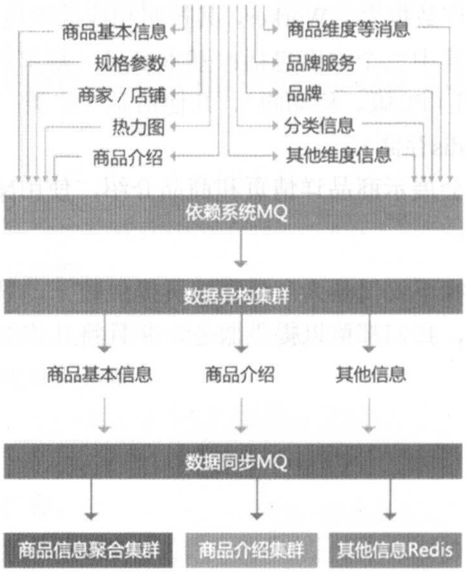
另外，我们的架构目标不仅仅是为商品详情页提供数据，只要是 Key-Value 获取数据，而非关系方式的数据，我们都可以提供服务，并且将其称之为动态服务系统。



该动态服务分为前端和后端，即公网还是内网，如目前该动态服务为列表页、商品对比、微信单品页、总代等提供相应的数据来满足和支持其业务。

16.6 详情页架构设计原则

16.6.1 数据闭环



数据闭环即数据的自我管理，或者说是数据都在自己系统里维护，不依赖于任何其他系统，即去依赖化这样的好处是别人抖动不会影响到我。数据闭环包括下面几个方面。

- 数据异构，这是数据闭环的第一步，即将各个依赖系统的数据拿过来，按照自己的要求存储起来。
- 数据原子化，数据异构的数据是原子化数据，这样未来我们可以对这些数据再加工再处理，从而响应变化的需求。
- 数据聚合，将多个原子数据聚合为一个大JSON数据，这样前端展示只需要一次获取，当然要考虑系统架构，比如我们使用的Redis改造，Redis又是单线程系统，我们需要部署更多的Redis来支持更高的并发，另外存储的值要尽可能小。
- 数据存储，我们使用JIMDB、Redis加持久化存储引擎，可以存储超过内存*N*倍的数据量。我们目前的一些系统使用的是Redis+LMDB引擎的存储，是配合SSD进行存储。另外，我们使用Hash Tag机制把相关的数据哈希到同一个分片，这样使用mget时不需要跨分片合并。

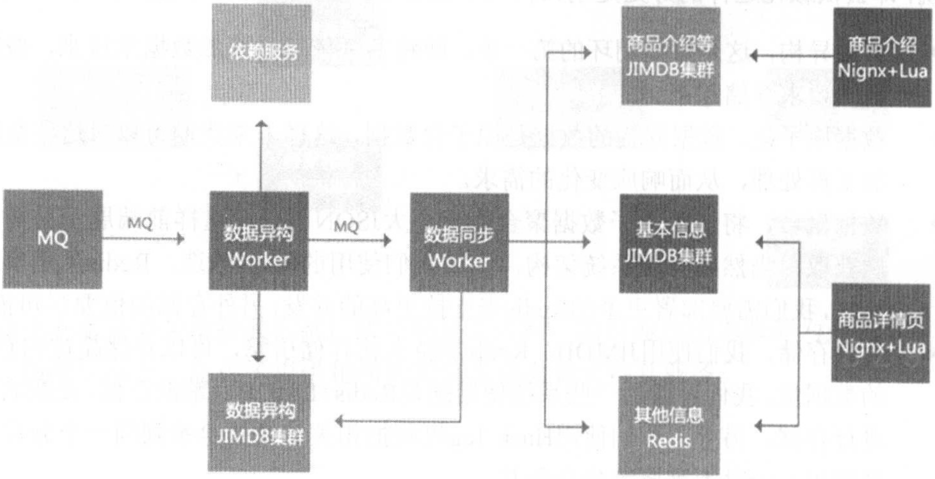
我们目前的异构数据是键值结构，用于按照商品维度查询，还有一套异构数据是关系结构的，用于关系查询。

16.6.2 数据维度化

数据应该按照维度和作用进行维度化，这样可以分离存储，进行更有效地存储和使用。示例数据的维度比较简单，如下。

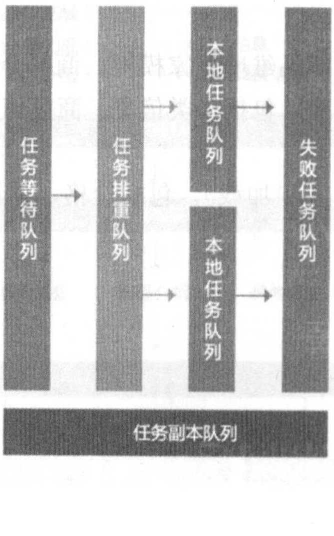
- 商品基本信息，包括标题、扩展属性、特殊属性、图片、颜色尺码、规格参数等。
- 商品介绍信息，包括商品维度商家模板、商品介绍等。
- 非商品维度的其他信息，包括分类信息、商家信息、店铺信息、店铺头、品牌信息等。
- 商品维度其他信息（异步加载），包括价格、促销、配送至、广告词、推荐配件、最佳组合等。

16.6.3 拆分系统



将系统拆分为多个子系统虽然增加了复杂性，但是可以得到更多的好处，比如数据异构系统存储的数据是原子化数据，这样可以按照一些维度对外提供服务；数据同步系统存储的是聚合数据，可以为前端展示提供高性能的读取；前端展示系统分离为商品详情页和商品介绍，可以减少相互影响。目前商品介绍系统还提供其他一些服务，比如全站异步页脚服务。

16.6.4 Worker 无状态化+任务化



- 对数据异构和数据同步Worker进行无状态化设计，这样可以水平扩展。
- 应用虽然是无状态化的，但是配置文件还是有状态的，每个机房一套配置，这样每个机房只读取当前机房数据。
- 任务多队列化，包括任务等待队列、包括任务排重队列、本地任务队列、失败任务队列。
- 队列优先级化，分为普通队列、刷数据队列、高优先级队列。比如，一些秒杀商品会用到高优先级队列，保证任务快速执行。
- 任务副本队列，当上线后业务出现问题时，修正逻辑可以回放，从而修复数据。可以按照诸如固定大小队列或者小时队列来进行设计。
- 在设计消息时，按照维度更新，比如商品信息变更和商品上下架分离，减少每次变更接口的调用量，通过聚合Worker去做聚合。

16.6.5 异步化+并发化

我们系统大量使用异步化，通过异步化机制提升并发能力。首先，我们使用了消息异步化进行系统解耦合，通过消息通知变更，然后再调用相应接口获取相关数据。之前老系统使用同步推送机制，这种方式下系统是紧耦合的，出问题时需要联系各个负责人重新推送，还要考虑失败重试机制。缓存数据更新异步化，同步调用服务，但异步更新缓存。让可并行任务并发化，虽然商品数据系统来源有多处，但是可以并发调用聚合，这样本来串行需要 1s 的任务，使用这种方式后只需要不到 300ms。异步请求做合并，然后一次请求调用就能拿到所有数据。前端服务异步化/聚合，实时价格、实时库存异步化，使用如线程或协程机制将多个可并发的服务聚合。异步化还一个好处就是可以对异步请求做合并，原来 N 次调用可以合并为一次，还可以做请求的排重。

16.6.6 多级缓存化

- **浏览器缓存：** 当页面之间来回跳转时走local cache，或者打开页面时拿着Last-Modified去CDN验证是否过期，这样可以减少来回传输的数据量。
- **CDN缓存：** 用户去离自己最近的CDN节点拿数据，而不是都回源到北京机房获取数据，这样可以提升访问性能。
- **服务器端应用本地缓存：** 我们使用Nginx+Lua架构，使用HttpLuaModule模块的shared dict做本地缓存（reload不丢失）或内存级Proxy Cache，从而减少带宽。

另外，我们还使用一致性哈希（如商品编号/分类）做负载均衡，内部对 URL 重写提升命中率。

我们对 mget 做了优化，如对于商品其他维度数据，分类、面包屑、商家等差不多 8 个维度数据，每次 mget 获取性能差而且数据量很大，基本在 30KB 以上。而这些数据缓存半小时也是没有问题的，因此我们设计为先读 local cache，然后把不命中的再回源到 remote cache 获取，这个优化减少了一半以上的 remote cache 流量。

服务器端分布式缓存，我们使用内存+SSD+JIMDB 持久化存储。

16.6.7 动态化

- **数据获取动态化：**商品详情页按维度获取数据，如商品基本数据、其他数据（分类、商家信息等）。而且可以根据数据属性按需做逻辑，比如虚拟商品需要自己定制详情页，那么我们就可以跳转，比如，全球购的需要走jd.hk域名，那么也是没有问题的。
- **模板渲染实时化：**支持随时变更模板需求。
- **重启应用秒级化：**使用Nginx+Lua架构，重启速度快，且不会丢共享字典缓存数据。
- **需求上线快速化：**因为我们使用了Nginx+Lua架构，因而可以快速上线和重启应用，不会产生抖动。另外，Lua本身是一种脚本语言，我们也在尝试把代码版本化存储，直接内部驱动Lua代码更新上线，而不需要重启Nginx。

16.6.8 弹性化

我们所有业务都使用 Docker 容器，但是数据库存储还是物理机。我们会制作一些基础镜像，把需要的软件打包成镜像，这样就不用每次去运维那安装部署软件了。未来可以支持自动扩容，比如，按照 CPU 或带宽自动扩容机器，目前京东的一些业务支持一分钟自动扩容。

16.6.9 降级开关

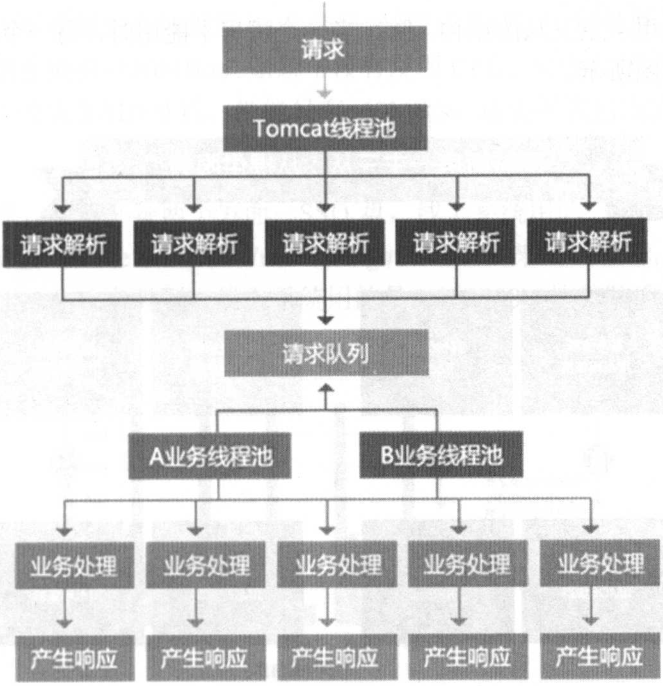
推送服务器推送降级开关，使开关集中化维护，然后通过推送机制推送到各个服务器。

可降级的多级读服务为前端数据集群→数据异构集群→动态服务（调用依赖系统）。这样可以保证服务质量，假设前端数据集群的一个磁盘坏了，还可以回源到数据异构集群获取数据。

将开关前置化，如 Nginx--àTomcat，在 Nginx 上做开关请求就到不了后端，减少后

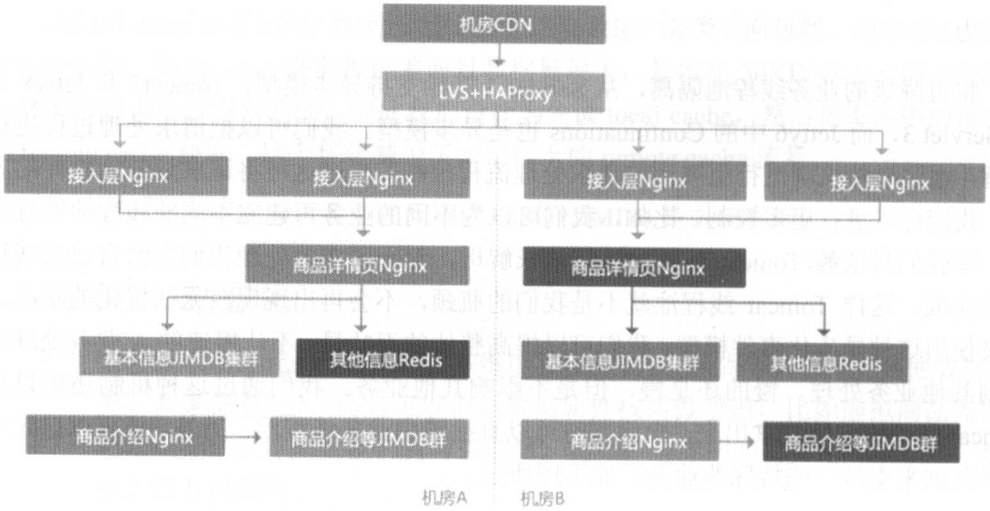
端压力。

将可降级的业务线程池隔离，从 Servlet 3 开始支持异步模型，Tomcat7 和 Jetty8 支持 Servlet 3，而 Jetty6 中的 Continuations 也是异步模型。我们可以把请求处理过程进行分解，通过事件机制进行更灵活的请求处理流程控制。通过这种将请求划分为事件的方式，我们可以进行更多控制。比如，我们可以为不同的业务再建立不同的线程池进行控制：即我们只依赖 Tomcat 线程池进行请求解析，对于请求的处理我们交给自己的线程池去完成。这样 Tomcat 线程池就不是我们的瓶颈，不会再出现现在无法优化的状况。通过使用这种异步化事件模型，我们可以提高整体的吞吐量，不让慢速的 A 业务处理影响到其他业务处理。慢的还是慢，但是不影响其他业务。我们通过这种机制还可以把 Tomcat 线程池的监控拿出来，出问题时可以直接清空业务线程池，另外，还可以自定义任务队列来支持一些特殊的业务，如下图所示。

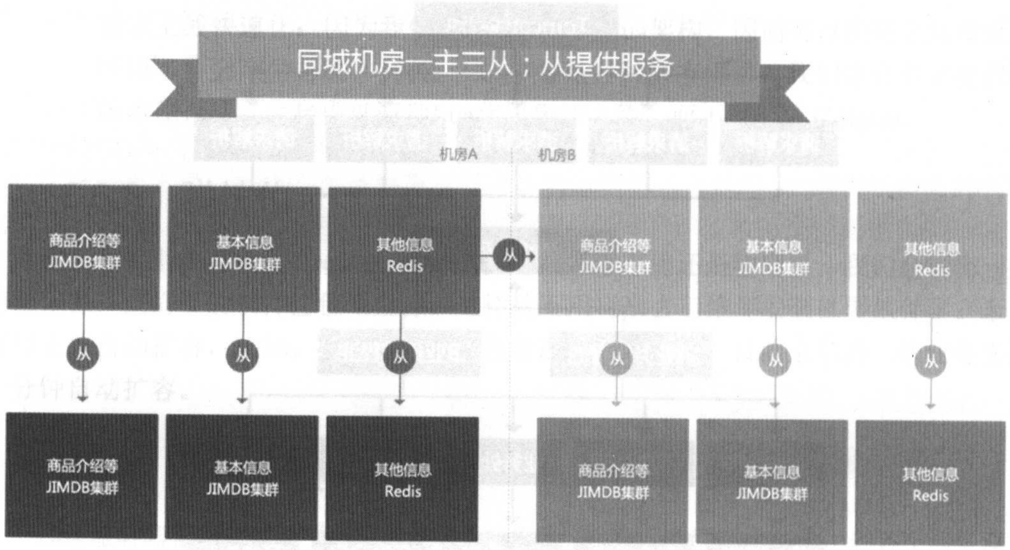


16.6.10 多机房多活

应用无状态，通过在配置文件中配置各自机房的数据集群来完成数据读取，如下页图所示。



数据集群采用一主三从的结构，防止当一个机房不能用时，另一个机房压力太大而产生抖动，如下图所示。



16.6.11 两种压测方案

第一种是线下压测，Apache ab、Apache Jmeter 这种方式是固定 URL 压测，一般通过访问日志收集一些 URL 进行压测，可以简单压测单机峰值吞吐量，但是，不能作为最终的压测结果，因为这种压测会存在热点问题。

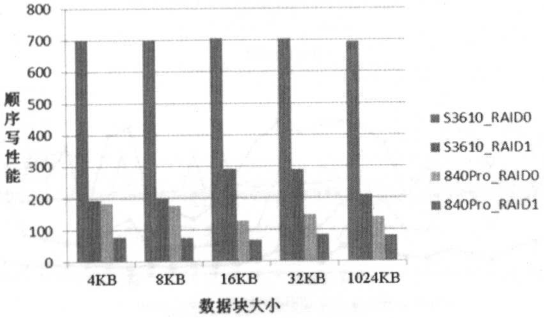
第二种是线上压测，可以使用 Tcpcopy 直接把线上流量导入到压测服务器，这种方式可以压测出机器的性能，而且可以把流量放大，也可以使用 Nginx+Lua 协程机制把流量分发到多台压测服务器，或者直接在页面埋点，让用户压测，此种压测方式可以不给用户返回内容。

16.7 遇到的一些坑和问题

16.7.1 SSD 性能差

使用 SSD 做 KV 存储时发现磁盘 IO 非常低。配置成 RAID 10 的性能只有 3~6MB/s。配置成 RAID 0 的性能有~130MB/s，系统中没有发现 CPU、MEM、中断等瓶颈。一台服务器从 RAID 1 改成 RAID 0 后，性能只有~60MB/s。这说明我们用的 SSD 盘性能不稳定。

根据以上现象，初步怀疑两个方面：SSD 盘，线上系统用的三星 840Pro 是消费级硬盘；RAID 卡设置，Write back 和 Write through 策略。后来测试验证，RAID 有影响，但不是关键。关于 RAID 卡类型，线上系统用的是 LSI 2008，比较陈旧。



本实验使用 dd 顺序写操作进行简单测试，严格测试需要用 FIO 等工具。

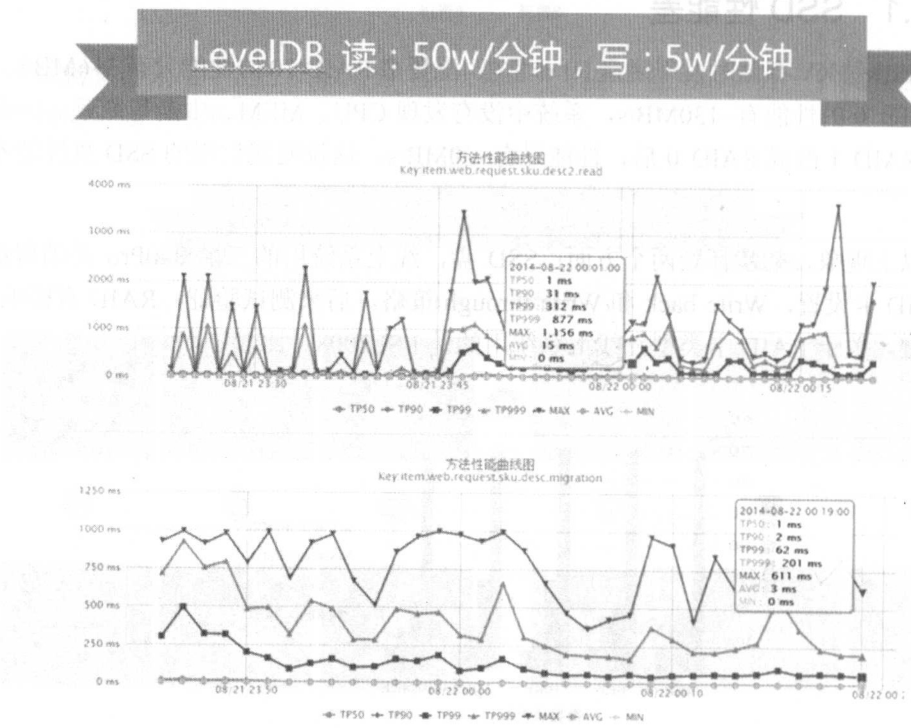
16.7.2 键值存储选型压测

我们在存储选型时尝试过 LevelDB、RocksDB、BeansDB、LMDB、Riak 等，最终

根据我们的需求选择了 LMDB。

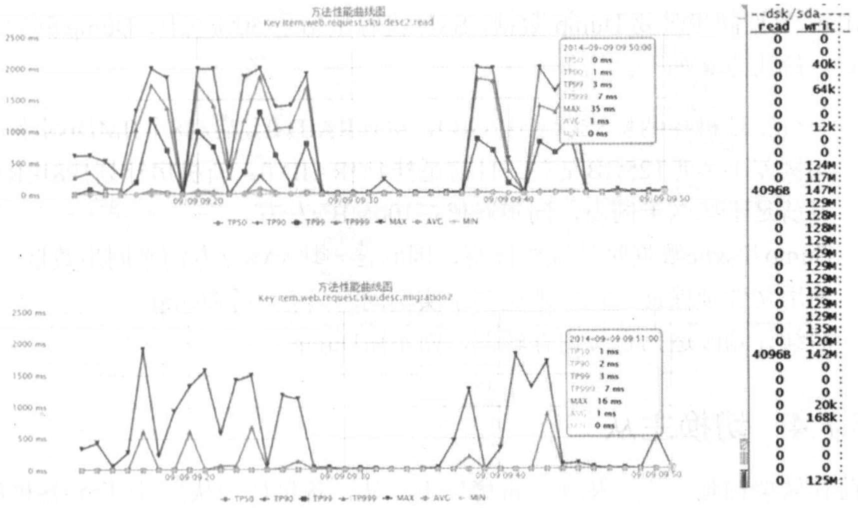
- **机器：** 两台
- **配置：** 32核CPU、32GB内存、SSD〔（512GB）三星840Pro →（600GB）Intel 3500 /Intel S3610〕
- **数据：** 1.7亿数据（超过800GB的数据）、大小5~30KB左右
- **KV存储引擎：** LevelDB、RocksDB、LMDB，每台启动两个实例
- **压测工具：** tcpcopy直接线上导流
- **压测用例：** 随机写+随机读

LevelDB 压测时，随机读+随机写会产生抖动（我们的数据出自自己的监控平台，分钟级采样），参见下图。



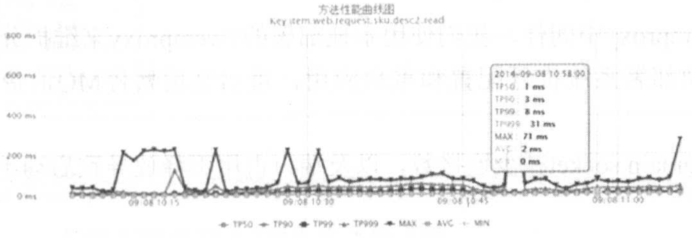
RocksDB 是改造自 LevelDB，对 SSD 做了优化，我们压测时采用单独写或读，性能非常好，但是读写混合时就会因为归并产生抖动，参见下页第一幅图。

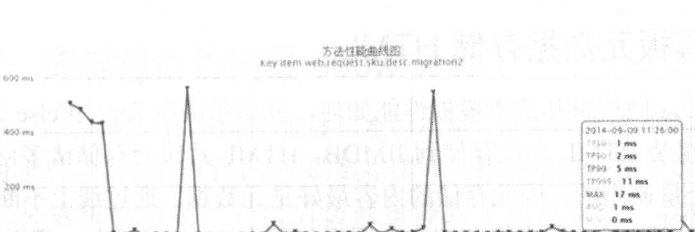
RocksDB 读：80w/分钟，写：2.3w/分钟



LMDB 引擎没有大的抖动，基本满足我们的需求，参见下图。

LMDB 读：80w/分钟，写：9w/分钟





目前线上我们的服务器使用的就是 LMDB。

16.7.3 数据量大时 JIMDB 同步不动

JIMDB 数据同步时要 Dump 数据，SSD 盘容量用了 50%以上，Dump 到同一块磁盘容量不足。解决方案如下。

- 一台物理机挂两块SSD（512GB），单挂RAID 0。启动8个JIMDB实例。这样每实例差不多是125GB左右。目前是挂4块RAID 0。新机房计划挂8块RAID 10。
- 目前是千兆网卡同步，同步峰值在100MB/s左右。
- Dump和sync数据时是顺序读写，因此挂一块SAS盘专门来同步数据。
- 使用文件锁保证一台物理机多个实例同时只有一个Dump。
- 后续计划改造为直接内存转发，而不做Dump。

16.7.4 切换主从

之前存储架构是一主二从（主机房一主一从，备机房一从），切换到备机房时，只有一台主服务器，读写压力大时有抖动，因此我们改造为之前架构图中的一主三从。

16.7.5 分片配置

之前的架构是分片逻辑分散到多个子系统的配置文件中，切换时需要操作很多系统，解决方案如下。

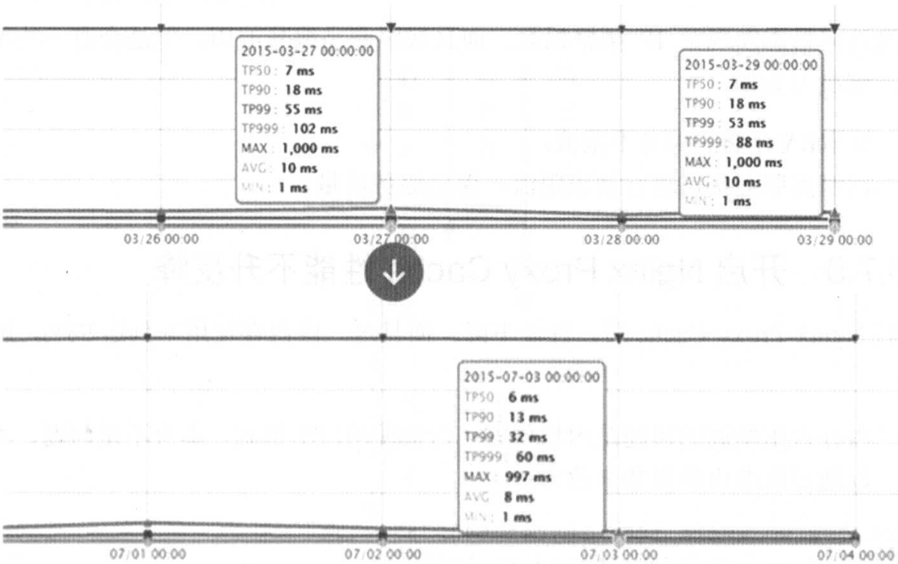
- 引入Twemproxy中间件，我们使用本地部署的Twemproxy来维护分片逻辑。
- 使用自动部署系统推送配置和重启应用，重启之前暂停MQ消费保证数据一致性。
- 用unix domain socket减少连接数，以及端口占用不释放导致启动不了服务的问题。

16.7.6 模板元数据存储 HTML

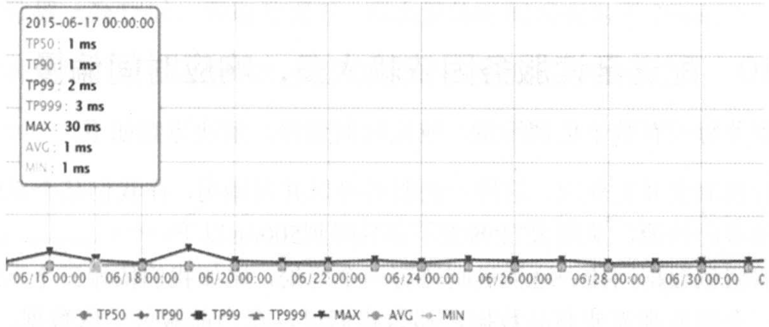
起初不确定 Lua 做逻辑和渲染模板性能如何，就尽量减少 for、if/else 之类的逻辑。通过 Java worker 组装 HTML 片段存储到 JIMDB，HTML 片段会存储诸多问题，假设未来变了也是需要全量刷出的，因此存储的内容最好是元数据。通过线上不断压测，最终 JIMDB 只存储元数据，Lua 做逻辑和渲染。逻辑代码在 3000 行以上。模板代码在 1500

行以上，其中有大量 for、if/else 语句，目前渲染性能可以接受。

线上真实流量，整体性能从 TP99 为 53ms 降到 TP99 为 32ms，参见下图。



绑定 8 CPU 测试的，渲染模板的性能可以接受，参见下图。



16.7.7 库存接口访问量 600w/分钟

商品详情页库存接口在 2014 年曾被恶意刷流量，每分钟超过 600w 访问量，Tomcat 机器只能定时重启。因为是详情页展示的数据缓存几秒钟是可以接受的，因此开启 Nginx Proxy Cache 来解决该问题，开启后访问量降到了正常水平。我们目前正在使用 Nginx+Lua 架构改造服务，数据过滤、URL 重写等在 Nginx 层完成，通过 URL 重写和

一致性哈希负载均衡，我们不再惧怕随机 URL，一些服务提升了 10%以上的缓存命中率。

16.7.8 微信接口调用量暴增

通过访问日志发现某 IP 频繁抓取。而且按照商品编号遍历，但是会有一些不存在的编号，解决方案如下。

- 读取KV存储的部分不限流。
- 回源到服务接口进行请求限流，保证服务质量。

16.7.9 开启 Nginx Proxy Cache 性能不升反降

开启 Nginx Proxy Cache 后，性能下降，而且过一段内存使用率到达 98%，解决方案如下。

- 内存占用率高的问题是内核问题，内核使用LRU机制，本身不是问题，不过可以通过修改内核参数来改善：

```
sysctl -w vm.extra_free_kbytes=6436787
sysctl -w vm.vfs_cache_pressure=10000
```

- 在HDD上使用Proxy Cache性能差，可以通过tmpfs缓存或Nginx共享字典缓存元数据，或者使用SSD，我们目前使用内存文件系统。

16.7.10 配送至读服务因依赖太多，响应时间偏慢

配送至服务每天有数十亿调用量，响应时间偏慢，解决方案如下。

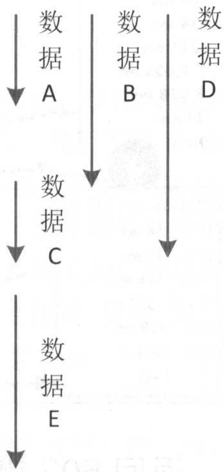
- 串行获取变并发获取，这样一些服务可以并发调用，在我们某个系统中能提升一倍多的性能，从原来TP99差不多1s降到500ms以下。
- 预取依赖数据回传，这种机制还有一个好处，比如我们依赖三个下游服务，而这三个服务都需要商品数据，那么我们可以在当前服务中取数据，然后回传，这样可以减少下游系统的商品服务调用量。如果没有传，那么下游服务再自己查一下。

数据如下表所示。

目标数据	数据 A	数据 B	数据 C	数据 D	数据 E
获取时间	10ms	15ms	20ms	5ms	10ms

如果串行获取，则需要 60ms。

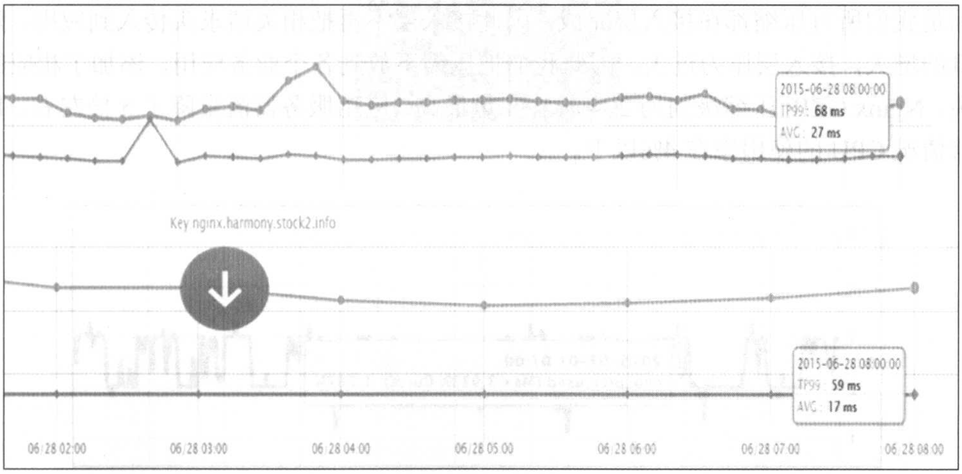
而如果数据 C 依赖数据 A 和数据 B，数据 D 谁也不依赖，数据 E 依赖数据 C，那么我们可以这样来获取数据：



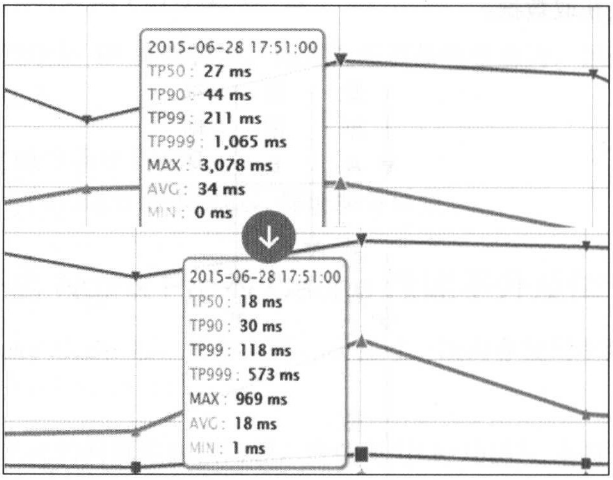
以这种获取数据只需要 30ms，能提升一倍的性能。

假设数据 E 还依赖数据 F（5ms），而数据 F 是在数据 E 服务中获取的，此时就可以考虑在获取数据 A/B/D 时，预取数据 F，那么整体时间就变为了 25ms。

通过这种优化，服务的整体性能的 TP99 差不多降低了 10ms。



如下服务是在抖动时的性能，老服务的 TP99 为 211ms，新服务的 TP99 为 118ms，此处我们主要就是并发调用+超时时间限制，超时直接降级。

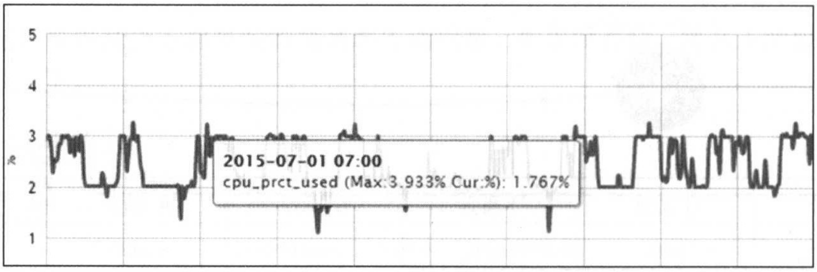


16.7.11 网络抖动时，返回 502 错误

Twemproxy 配置的 timeout 时间太长，之前设置为 5s，而且没有分别针对连接、读、写设置超时。后来我们减少超时时间，内网设置在 150ms 以内，当超时时访问动态服务。

16.7.12 机器流量太大

2014 年“双 11”期间，服务器网卡流量到了 400Mbps，CPU 的使用率在 30%左右。原因是我们所有压缩都在接入层完成，因此接入层不再把相关请求头传入到应用，随着流量的增大，接入层压力过大，因此我们把压缩下放到各个业务应用，添加了相应的请求头，Nginx GZIP 压缩级别为 2~4 时吞吐量最高。应用服务器流量降了 5 倍左右。目前正常情况 CPU 的使用率在 4%以下。



16.8 其他

在 Nginx 接入层实现线上灰度引流。在接入层转发请求时只保留有用请求头，因而后端 Tomcat 不必解析无用的请求头。使用不带 Cookie 的无状态域名（如 c.3.cn）减少请求流量，比如 jd.com 域名下边可能存在 1KB 的 Cookie，但是服务器端根本不需要。Nginx Proxy Cache 只缓存有效数据，如托底数据不缓存，避免缓存一些错误的数据。使用非阻塞锁应对本地缓存失效时突发请求到后端应用（lua-resty-lock/proxy_cache_lock）。使用 Twemproxy 等代理减少 Redis 连接数。使用 unix domain socket 套接字减少本机 TCP 连接数。设置合理的超时时间（连接、读、写）。使用长连接减少内部服务的连接数。去数据库依赖（协调部门迁移数据库是很痛苦的，目前内部使用机房域名而不是 IP 地址）。客户端同域连接限制，进行域名分区：c0.3.cn、c1.3.cn，如果未来支持 HTTP/2.0 的话，则不再适用。

想要了解京东手机详情页架构，可扫二维码参考《京东手机商品详情页技术解密》。

17 京东商品详情页服务闭环实践

京东商品详情页技术方案在第 16 章已经详细介绍了，接下来为大家揭秘双 11 抗下几十亿流量的商品详情页统一服务架构，这次双 11 整个商品详情页没有出现不服务的情况，服务非常稳定。统一服务提供了促销和广告词合并服务、库存状态/配送至服务、延保服务、试用服务、推荐服务、图书相关服务、详情页优惠券服务、今日抄底服务等服务支持。这些服务中有我们自己做的服务实现，还有一些是简单做一下代理或者接口，做合并输出到页面，我们将这些服务聚合到一个系统的目的是打造服务闭环，优化现有服务，并为未来需求做准备，跟着自己的方向走，而不被别人打乱我们的方向。

大家在页面中看到的 c.3.cn/c0.3.cn/c1.3.cn/cd.jd.com 请求都是统一服务的入口。

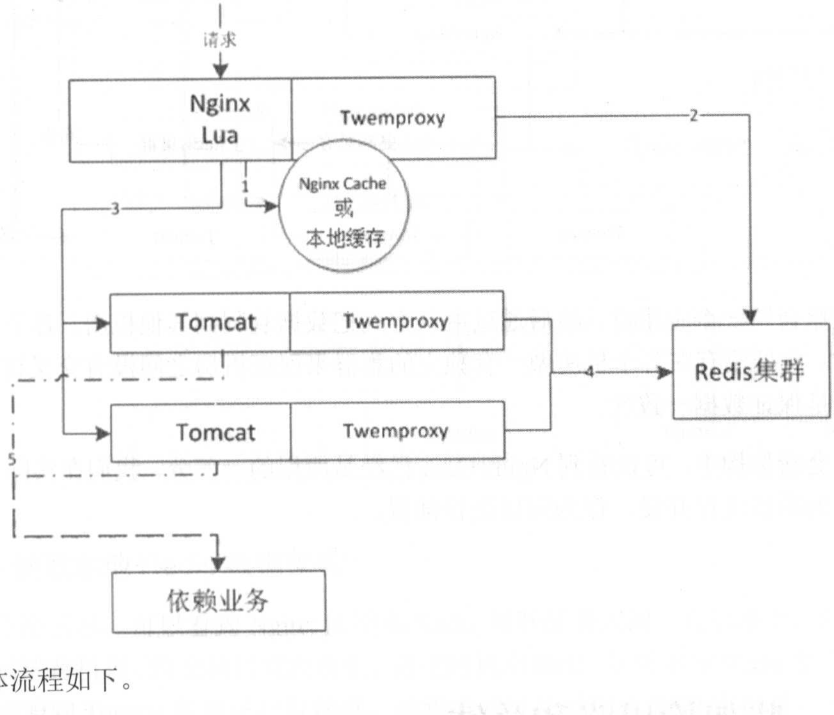
17.1 为什么需要统一服务

商品详情页虽然只有一个页面，但是依赖的服务众多，我们需要把控好入口，统一化管理。这样的好处是在统一管理和监控下，出问题可以统一降级。可以把一些相关接口合并输出，减少页面的异步加载请求。一些前端逻辑后移到服务器端，前端只做展示，

不进行逻辑处理。

有了它，所有入口都在我们的服务中，我们可以更好地监控和思考我们页面的服务，让我们能运筹于帷幄之中，决胜于千里之外。在设计一个高度灵活的系统时，要想着当出现问题时怎么办：是否可降级？不可降级怎么处理？是否会发送滚雪球问题？如何快速响应异常？完成了系统核心逻辑只能保证服务能工作，服务如何更好更有效或者在异常情况下正常工作，也是我们要深入思考和解决的问题。

17.2 整体架构



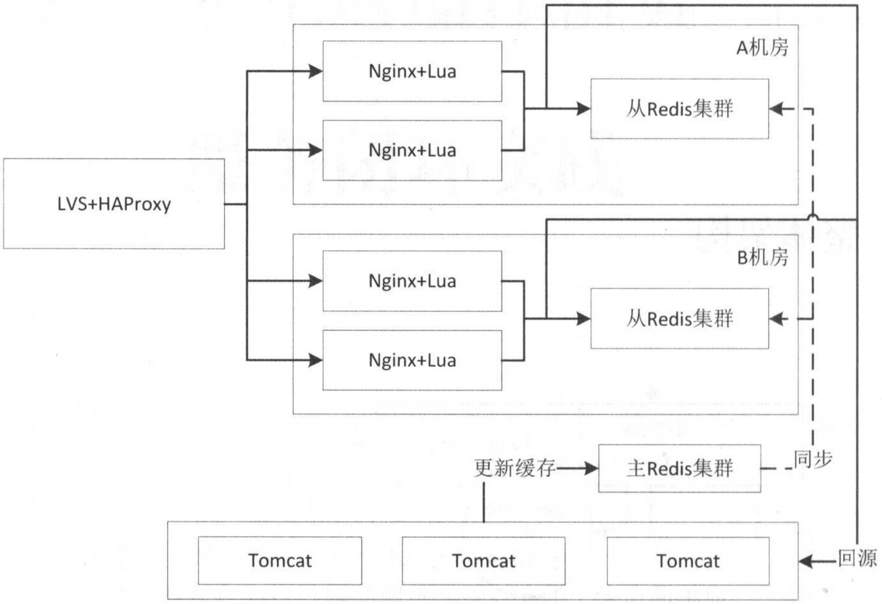
整体流程如下。

1．请求首先进入 Nginx，Nginx 调用 Lua 进行一些前置逻辑处理，如果前置逻辑不合法，那么直接返回，然后查询本地缓存，如果命中，则直接返回数据。

2．如果本地缓存未命中数据，则查询分布式 Redis 集群；如果命中数据，则直接返回。

3．如果分布式 Redis 集群未命中数据，则调用 Tomcat 进行回源处理。然后把结果异步写入 Redis 集群，并返回。

如上是整个逻辑流程，可以看到我们在 Nginx 这一层做了很多前置逻辑处理，以此来减少后端压力，另外，我们的 Redis 集群分机房部署如下图所示。



即数据会写一个主集群，然后通过主从方式把数据复制到其他机房，各个机房读自己的集群。此处没有在各个机房做一套独立的集群来保证机房之间没有交叉访问，这样做的目的是保证数据一致性。

在这套新架构中，可以看到 Nginx+Lua 已经是应用的一部分，我们在实际使用中也是把它作为项目进行开发，作为应用进行部署。

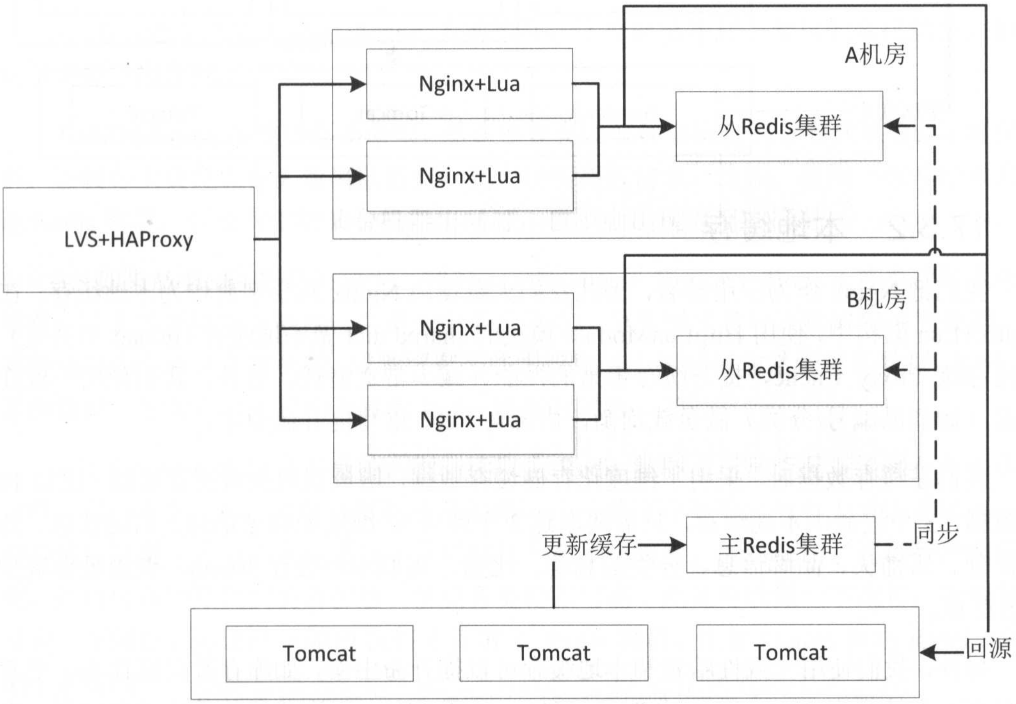
17.3 一些架构思路和总结

我们主要遵循如下几个原则设计系统架构。

17.3.1　两种读服务架构模式

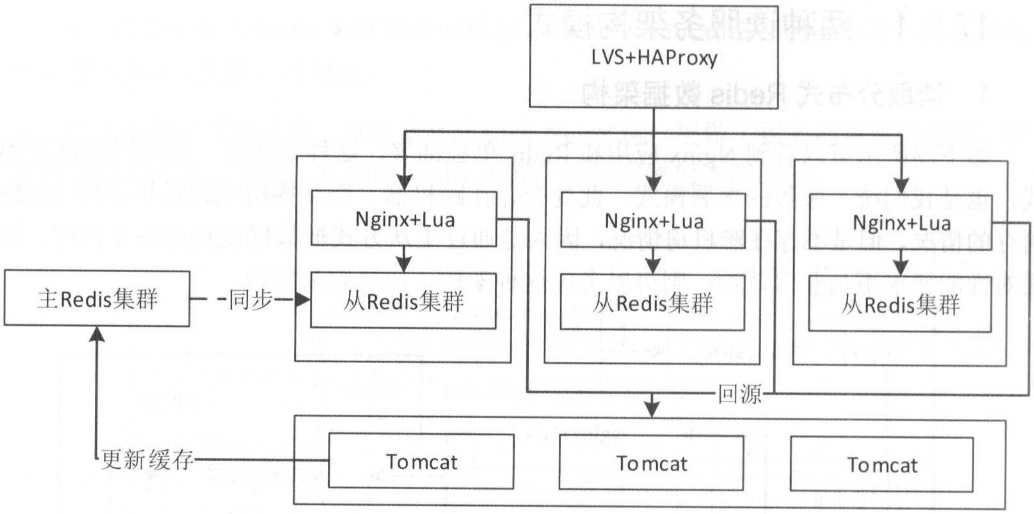
1. 读取分布式 Redis 数据架构

如下图所示可以看到 Nginx 应用和 Redis 单独部署，这种方式是一般应用的部署模式，也是我们统一服务的部署模式，此处会存在跨机器、跨交换机或跨机柜读取 Redis 缓存的情况，但是不存在跨机房情况，因为是通过主从方式把数据复制到各个机房。如果对性能要求不是非常苛刻，则可以考虑这种架构，比较容易维护。



2. 读取本地 Redis 数据架构

如下图所示，可以看到 Nginx 应用和 Redis 集群部署在同一台机器上，这样的好处是可以消除跨机器、跨交换机或跨机柜，甚至跨机房调用。如果本地 Redis 集群不命中，则还是回源到 Tomcat 集群进行取数据。此种方式可能受限于 TCP 连接数，可以考虑使用 unix domain socket 套接字减少本机 TCP 连接数。如果单机内存成为瓶颈（比如单机内存最大为 256GB），那么就需要路由机制来进行分片，比如，按照商品尾号分片，Redis 集群一般采用树状结构挂主从部署。



17.3.2 本地缓存

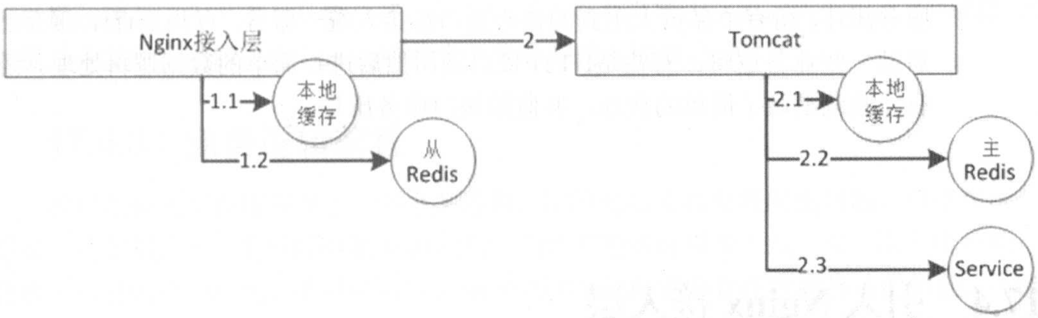
我们把 Nginx 作为应用部署，因此我们大量使用 Nginx 共享字典作为本地缓存。在 Nginx+Lua 架构中，使用 HttpLuaModule 模块的 shared dict 做本地缓存（reload 不丢失）或内存级 Proxy Cache，提升缓存带来的性能并减少带宽消耗。另外，我们使用一致性哈希（如商品编号/分类）做负载均衡，内部对 URL 重写提升命中率。

我们在缓存数据时，采用了维度化存储缓存数据，增量获取失效缓存数据（比如 10 个数据，3 个没命中本地缓存，只需要取这 3 个即可）。维度如商家信息、店铺信息、商家评分、店铺头、品牌信息、分类信息等。比如，我们本地缓存 30min，调用量会减少 3 倍左右。

另外，我们使用一致性哈希和本地缓存可以提升命中率，如库存数据缓存 5s，平常命中率：本地缓存 25%，分布式 Redis 28%，回源 47%；一次普通秒杀活动命中率：本地缓存 58%、分布式 Redis 15%，回源 27%，而某个服务使用一致哈希后命中率提升 10%；防止 URL 顺序不同导致缓存命中率低，或存在一些如时间这种随机参数，即页面 URL 不管怎么变都不要让它成为缓存不命中的因素。

17.3.3 多级缓存

对于读服务，我们在设计时会使用多级缓存来尽量减少后端服务压力，在统一服务系统中，我们设计了 4 级缓存，如下图所示。



1．首先在接入层会使用 Nginx 本地缓存，这种前端缓存主要目的是抗热点，根据场景来设置缓存时间。

2．如果 Nginx 本地缓存不命中，那么接着会读取各个机房的分布式从 Redis 缓存集群，该缓存主要是保存大量离散数据，抗大规模离散请求，比如，使用一致性哈希来构建 Redis 集群，即使其中的某台机器出问题，也不会出现雪崩的情况。

3．如果从 Redis 集群不命中，则 Nginx 会回源到 Tomcat。Tomcat 首先读取本地堆缓存，这个主要用来支持在一个请求中多次读取一个数据或者与该数据相关的数据。而其他情况命中率是非常低的，或者缓存一些规模比较小但用得非常频繁的数据，如分类、品牌数据。堆缓存时间我们设置为 Redis 缓存时间的一半。

4．如果 Java 堆缓存不命中，则会读取主 Redis 集群，正常情况下该缓存命中率非常低，只有不到 5%。读取该缓存的目的是防止前端缓存失效之后有大量请求涌入，进而导致后端服务压力太大而雪崩。我们默认开启了该缓存，虽然增加了几毫秒的响应时间，但是可以加厚我们的防护盾，使服务更稳当可靠。此处可以做一下改善，比如我们设置一个阈值，超过这个阈值我们才读取主 Redis 集群，比如 Guava 就由 RateLimiter API 来实现。

17.3.4 统一入口/服务闭环

在第 16 章中已经讲过了数据异构闭环的收益，在统一服务中我们也遵循这个设计原则，此处我们主要做了两件事情。

- 数据异构，如我们对判断库存状态依赖的套装、配件关系进行了异构，未来可以对商家运费等数据进行异构，减少接口依赖。

- 服务闭环，所有单品页上用到的核心接口都接入统一服务。有些是查库/缓存然后做一些业务逻辑，有些是HTTP接口调用然后进行简单的数据逻辑处理。还有一些就是做了简单的代理，并监控接口服务质量。

17.4 引入 Nginx 接入层

我们在设计系统时需要把一些逻辑尽可能前置，以此来减轻后端核心逻辑的压力，而且可以让服务升级/服务降级非常方便地进行切换，在接入层我们做了如下事情。

17.4.1 数据校验/过滤逻辑前置

我们的服务有两种类型的接口：一种是与用户无关的接口，另一种则是与用户相关的接口。因此我们使用了两种类型的域名 c.3.cn/c0.3.cn/c1.3.cn 和 cd.jd.com。当我们请求 cd.jd.com 时会带着用户 Cookie 信息到服务器端。在服务器上会进行请求头的处理，用户无关的所有数据通过参数传递，在接入层会丢弃所有的请求头（保留 gzip 相关的头）。而用户相关的数据会从 Cookie 中解出用户信息，然后通过参数传递到后端。也就是后端应用从来就不关心请求头及 Cookie 信息，所有信息通过参数传递。

请求进入接入层后，会对参数进行校验，如果参数校验不合法，则直接拒绝这次请求。我们对每个请求的参数进行了最严格的数据校验处理，保证数据的有效性。如下图所示，我们对关键参数进行了过滤，如果这些参数不合法，那么就直接拒绝请求。

```
local skuId = getSkuId(args['skuId']) --sku
local venderId = getVenderId(args['venderId']) --商家id
local cat = getCat(args['cat'])  --分类
local area = getArea(args['area']) --区域
```

另外，我们还会对请求的参数进行过滤，然后按照固定的模式重新拼装 URL 调度到后端应用，此时，URL 上的参数是固定的而且是有序的，可以按照 URL 进行缓存。

17.4.2 缓存前置

我们把很多缓存前置到了接入层来进行热点数据的削峰，而且配合一致性哈希也许

可以提升缓存的命中率。在缓存时，我们按照业务来设置缓存池，减少相互之间的影响并提升并发率。我们使用 Lua 读取共享字典来实现本地缓存。

17.4.3 业务逻辑前置

我们在接入层直接实现了一些业务逻辑，原因是如果在高峰时出问题，可以在这一层做一些逻辑升级。我们后端是 Java 应用，当修复逻辑时需要上线，而一次上线可能花费数十秒时间启动应用，重启应用后 Java 应用 JIT 的问题会存在性能抖动的问题，可能因为重启造成服务一直启动不起来的问题。而在 Nginx 中做这件事情，改完代码推送到服务器，重启只需要秒级，而且不存在抖动问题，这些逻辑都是在 Lua 中完成的。

17.4.4 降级开关前置

我们将降级开关分为这么几种：接入层开关和后端应用开关、总开关和原子开关。我们在接入层设置开关的目的是为了防止降级后流量还无谓地打到后端应用。总开关是对整个服务降级，比如，库存服务默认有货。而原子开关是对整个服务中的一个小服务降级，比如库存服务中需要调用商家运费服务，如果只是商家运费服务出问题了，则此时可以只降级商家运费服务。另外，我们还可以根据服务重要程度来使用超时自动降级机制。

我们使用 init_by_lua_file 初始化开关数据，共享字典存储开关数据，提供 API 进行开关切换（switch_get(“stock.api.not.call”) ~= “1”）。可以实现秒级切换开关、增量式切换开关（可以按照机器组开启，而不是所有都开启）、功能切换开关、细粒度服务降级开关，非核心服务可以实现超时自动降级。

比如，双 11 期间有些服务出问题了，我们进行过大服务和小服务的降级操作，这些操作对用户来说都是无感知的。

17.4.5 A/B 测试

对于服务升级，最重要的就是做 A/B 测试，然后根据 A/B 测试的结果来看是否切换新服务。而有了接入层非常容易进行这种 A/B 测试。不管是上线还是切换都非常容易。可以在 Lua 中根据请求的信息调用不同的服务，或者通过 upstream 分组即可完成 A/B 测试。

17.4.6 灰度发布/流量切换

对于一个灵活的系统来说，能随时进行灰度发布和流量切换是非常重要的一件事情，比如，验证新服务器是否稳定，或者验证新的架构是否比老架构更优秀，有时只有在线上运行才能看出是否有问题。我们在接入层可以通过配置或者写 Lua 代码来完成这件事情，灵活性非常好。可以设置多个 upstream 分组，然后根据需要切换分组即可。

17.4.7 监控服务质量

对于一个系统来说，最重要的是要有一双眼睛能盯着系统，以便尽可能早地发现问题，我们在接入层会对请求进行代理，记录 status、request_time、response_time 来监控服务质量，比如，根据调用量、状态码是否是 200、响应时间来告警。

17.4.8 限流

我们系统中存在的主要限流逻辑是，对于大多数请求按照 IP 请求数限流，对于登录用户按照用户限流。对于读取缓存的请求不进行限流，只对打到后端系统的请求进行限流。还可以限制用户访问频率，比如，使用 ngx_lua 中的 ngx.sleep 对请求进行休眠处理，让刷接口的速度降下来。或者种植 cookie token 之类的，必须按照流程访问。当然还可以对爬虫/刷数据的请求返回假数据来减少影响。

17.5 前端业务逻辑后置

前端 JS 应该尽可能少写业务逻辑和一些切换逻辑，因为前端 JS 一般推送到 CDN，假设逻辑出问题了，需要更新代码上线，推送到 CDN 然后让各个边缘 CDN 节点失效，或者通过版本号机制在服务器端模板中修改版本号上线。这两种方式都存在效率问题，假设处理一个紧急故障，用这两种方式处理的过程中可能故障早已经恢复了。因此，我们的观点是前端 JS 只拿数据展示，所有或大部分逻辑交给后端去完成，即静态资源 CSS/JS CDN，动态资源 JSONP。前端 JS 瘦身，业务逻辑后置。

在“双 11”期间我们的某些服务出问题了，不能更新商品信息，此时秒杀商品需要打标处理，因此我们在服务器端完成了这件事情，整个处理过程只需要几十秒就能搞定，

避免了商品不能被秒杀的问题。而如果在 JS 中完成，则需要耗费非常长的时间，因为 JS 在客户端还有缓存时间，而且一般缓存时间非常长。

17.6 前端接口服务器端聚合

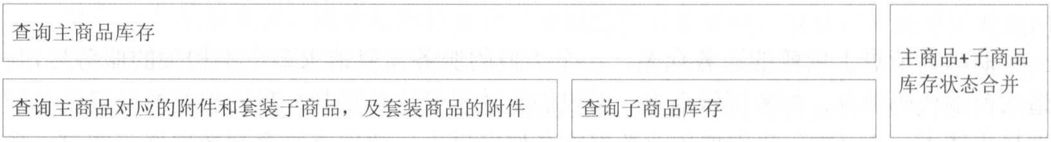
商品详情页上依赖的服务众多，一个类似的服务需要请求多个不相关的服务接口，造成前端代码臃肿，判断逻辑众多。而我们无法忍受这种现状，我们想要的结果就是前端异步请求一个 API，我们把相关数据准备好发过去，前端直接拿到数据展示即可。所有或大部分逻辑在服务器端完成而不是在客户端完成。因此，我们在接入层使用 Lua 协程机制并发调用多个相关服务，最后把这些服务进行合并，比如几种推荐服务：最佳组合、推荐配件、优惠套装。通过 http://c.3.cn/recommend?methods=accessories,suit,combination&sku=1159330&cat=6728,6740,12408&lid=1&lim=6 进行请求获取聚合的数据，这样原来前端需要调用三次的接口只需要一次就能吐出所有数据。

我们对这种请求进行了 API 封装，如下图所示。

```
local apis = {}
if switch_get("accessories.api.not.call") ~= "1" and methods['accessories'] and checkIsEnableAccess(skuId, cat[1], cat[2]) then
    local area = args.area or ""
    apis['accessories'] = {
        method = 'accessories',
        callback = nil,
        charset = "gbk",
        url = "/recommend/accessories",
        args = "skuId=" .. skuId .. "&c1=" .. cat[1] .. "&c2=" .. cat[2] .. "&c3=" .. cat[3] .. "&area=" .. area
    }
end

if switch_get("suit.api.not.call") ~= "1" and methods['suit'] then
    apis['suit'] = {
        method = 'suit',
        callback = nil,
        charset = "gbk",
        url = "/recommend/suit",
        args = "skuId=" .. skuId
    }
end

invokeApis(apis, true)
```

比如库存服务，商品是否有货需要判断：主商品库存状态、主商品对应的套装、子商品库存状态、主商品附件库存状态及套装子商品附件库存状态。套装商品是一个虚拟

商品，是多个商品绑定在一起进行售卖的形式。如果这段逻辑放在前段完成，则需要多次调用库存服务，然后进行组合判断，这样前端代码会非常复杂，凡是涉及调用库存的服务都要进行这种判断。因此，我们把这些逻辑封装到服务端完成。前端请求 http://c0.3.cn/stock?skuId=1856581&venderId=0&cat=9987,653,655&area=1_72_2840_0&buyNum=1&extraParam={%22originid%22:%221%22}&ch=1&callback=getStockCallback，然后服务端计算整个库存状态，而前端不需要做任何调整。在服务端使用 Lua 协程并发的进行库存调用，如下图所示。



再比如今日抄底服务，调用接口太多，如库存、价格、促销等都需要调用，因此，我们也使用这种机制把这几个服务在接入层合并为一个大服务，对外暴露。http://c.3.cn/today?skuId=1264537&area=1_72_2840_0&promotionId=182369342&cat=737,752,760&callback=jQuery9364459&_=1444305642364。

我们目前合并的主要有：促销和广告词合并、配送至相关服务合并。而未来这些服务都会合并，并会在前端进行一些特殊处理，比如设置超时，超时后自动调用原子接口。接口吐出的数据状态码不对，再请求一次原子接口获取相关数据。

17.7 服务隔离

服务隔离的目的是防止因为某些服务抖动而造成整个应用内的所有服务不可用，可以分为应用内线程池隔离、部署/分组隔离、拆应用隔离。

- **应用内线程池隔离：**我们采用了Servlet 3异步化，并为不同的请求按照重要级别分配线程池，这些线程池是相互隔离的，我们也提供了监控接口以便发现问题并及时进行动态调整，该实践可以参考第3章内容。

- **部署/分组隔离：**意思是为不同的消费方提供不同的分组，不同的分组之间相互不影响，以免因为大家使用同一个分组导致有些人乱用，致使整个分组服务不可用。
- **拆应用隔离：**如果一个服务调用量巨大，那么我们便可以把这个服务单独拆出去，做成一个应用，减少因其他服务上线或者重启导致影响本应用。

18 使用OpenResty开发高性能Web应用

在互联网公司，Nginx 可以说是标配组件，但是主要场景还是负载均衡、反向代理、代理缓存、限流等。而把 Nginx 作为一个 Web 容器使用的还不是那么广泛。Nginx 的高性能是大家公认的，而 Nginx 开发主要是以 C/C++模块的形式进行，整体学习和开发成本偏高。如果需要一种简单的语言来实现 Web 应用的开发，那么 Nginx 绝对是个好选择。目前，Nginx 团队也开始意识到这个问题，开发了 nginxScript，可以在 Nginx 中使用 JavaScript 进行动态配置一些变量和动态脚本执行。而目前市面上用得非常成熟的扩展是由章亦春将 Lua 和 Nginx 黏合的 ngx_lua 模块，并且将 Nginx 核心、LuaJIT、ngx_lua 模块、许多有用的 Lua 库和常用的第三方 Nginx 模块组合在一起成为 OpenResty，这样开发人员就可以安装 OpenResty，使用 Lua 编写脚本，然后部署到 Nginx Web 容器中运行，这样就能非常轻松地开发出高性能的 Web 服务。

接下来，我们就认识一下构成 Open Resty 的 Nginx、Lua、ngx_lua 模块，以及 OpenResty 到底能开发哪些类型的 Web 应用。

18.1 OpenResty 简介

18.1.1 Nginx 优点

Nginx 设计为一个主进程多个工作进程的工作模式，每个进程是单线程来处理多个连接，而且每个工作进程采用了非阻塞 I/O 来处理多个连接，从而减少了线程上下文切换，实现了公认的高性能、高并发。因此，在生成环境中会通过把 CPU 绑定给 Nginx 工作进程，从而提升其性能。另外，因为单线程工作模式的特点，内存占用就非常少了。

Nginx 更改配置后重启速度非常快，可以达到毫秒级，而且支持不停止 Nginx 进行升级 Nginx 版本、动态重载 Nginx 配置。

Nginx 模块也非常多，功能也很强劲，不仅可以作为 HTTP 负载均衡，Nginx 发布 1.9.0 版本还支持 TCP 负载均衡，还可以很容易地实现内容缓存、Web 服务器、反向代理、访问控制等功能。

18.1.2 Lua 的优点

Lua 是一种轻量级、可嵌入式的脚本语言，可以非常容易地嵌入到其他语言中使用。另外，Lua 提供了协程并发，即以同步调用的方式进行异步执行，从而实现并发，比起回调机制的并发来说代码更容易编写和理解，排查问题也会更容易。Lua 还提供了闭包机制，函数可以作为 First Class Value 进行参数传递，另外，其实现了标记清除垃圾收集。

因为 Lua 的小巧轻量级，可以在 Nginx 中嵌入 Lua VM，请求的时候创建一个 VM，请求结束的时候回收 VM。

18.1.3 什么是 ngx_lua

ngx_lua 是章亦春编写的 Nginx 的一个模块，将 Lua 嵌入到 Nginx 中，从而可以使用 Lua 来编写脚本，部署到 Nginx 中运行，即 Nginx 变成了一个 Web 容器。这样开发人员就可以使用 Lua 语言开发高性能 Web 应用了。

ngx_lua 提供了与 Nginx 交互的很多 API，对于开发人员来说只需要学习这些 API 就可以进行功能开发，而对于开发 Web 应用来说，如果接触过 Servlet 的话，你会发现其开发和 Servlet 类似，无外乎就是知道接收请求、参数解析、功能处理、返回响应这几步的 API 是什么样子的。

18.1.4 开发环境

我们使用 OpenResty 来搭建开发环境，OpenResty 将 Nginx 核心、LuaJIT、许多有用的 Lua 库和 Nginx 第三方模块打包在一起。这样开发人员只需要安装 OpenResty，不需要了解 Nginx 核心和写复杂的 C/C++模块，只需要使用 Lua 语言进行 Web 应用开发。

如何安装可扫描下面二维码参考笔者写的《跟我学 OpenResty（Nginx+Lua）开发》教程。

18.1.5 OpenResty 生态

OpenResty 提供了一些常用的 ngx_lua 开发模块，如 lua-resty-memcached、lua-resty-mysql、lua-resty-redis、lua-resty-dns、lua-resty-limit-traffic、lua-resty-template。

这些模块涉及如 MySQL 数据库、Redis、限流、模块渲染等常用功能组件。另外，也有很多第三方的 ngx_lua 组件供我们使用，对于大部分应用场景来说，现在生态环境中的组件已经足够多了。如果不满足需求，那么也可以自己去写来完成需求。

18.1.6 场景

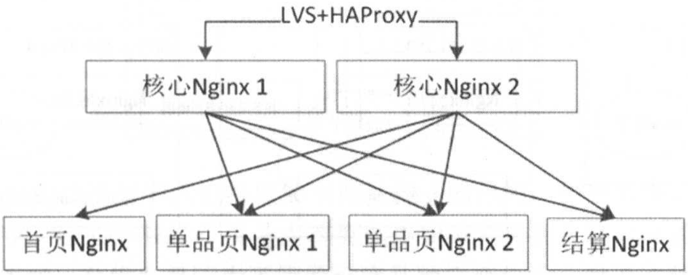
目前 CDN 厂商使用 OpenResty 较多，而像京东有使用 OpenResty 开发复杂的 Web 应用。目前见到的一些应用场景如下。

- **Web应用**：会进行一些业务逻辑处理，甚至进行耗CPU的模板渲染，一般流程包括mysql/Redis/HTTP获取数据、业务处理、产生JSON/XML/模板渲染内容，比如，京东的列表页/商品详情页。
- **接入网关**：实现如数据校验前置、缓存前置、数据过滤、API请求聚合、A/B测试、灰度发布、降级、监控等功能，比如，京东的交易大Nginx节点、无线部门正在开发的无线网关、单品页统一服务、实时价格、动态服务。
- **Web防火墙**：可以进行IP/URL/UserAgent/Referer黑名单、限流等功能。
- **缓存服务器**：可以对响应内容进行缓存，减少到达后端的请求，从而提升性能。
- **其他**：如静态资源服务器、消息推送服务、缩略图裁剪等。

18.2 基于 OpenResty 的常用架构模式

18.2.1 负载均衡

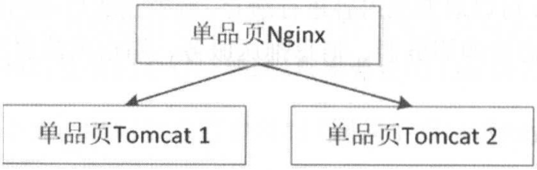
如下图所示，我们首先通过 LVS+HAProxy 将流量转发给核心 Nginx 1 和核心 Nginx 2，即实现了流量的负载均衡，此处可以使用如轮询、一致性哈希等调度算法来实现负载的转发。然后核心 Nginx 会根据请求特征如“Host:item.jd.com”，转发给相应的业务 Nginx 节点，如单品页 Nginx 1。此处为什么分两层呢？



- 核心Nginx层是无状态的，可以在这一层实现流量分组（内网和外网隔离、爬虫和非爬虫流量隔离）、内容缓存、请求头过滤、故障切换（机房故障切换到其他机房）、限流、防火墙等一些通用型功能。

- 业务Nginx，如单品页Nginx，可以在业务Nginx实现业务逻辑，或者反向代理到如Tomcat，在这一层可以实现内容压缩（放在这一层的目的是减少核心Nginx的CPU压力，将压力分散到各业务Nginx）、A/B测试、降级。即这一层的Nginx跟业务有关联，实现业务的一些通用逻辑。

不管是核心 Nginx 还是业务 Nginx，都应该是无状态设计，可以水平扩容，如下图所示。

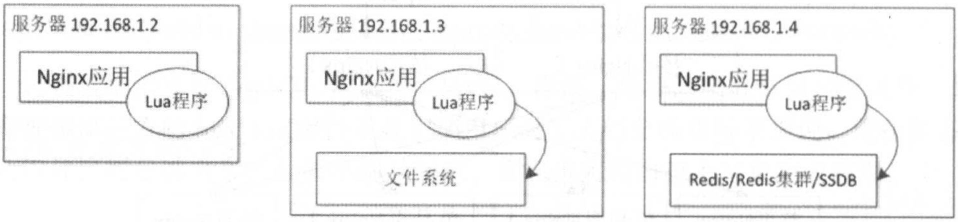


业务 Nginx 一般会把请求直接转发给后端的业务应用，如 Tomcat、PHP，即将请求内部转发到相应的业务应用。当有的 Tomcat 出现问题时，可以在这一层摘掉。或者有的业务路径变了在这一层进行重写。或者有的后端 Tomcat 压力太大也可以在这一层降级，减少对后端冲击。或者业务需要灰度发布时，也可以在这一层 Nginx 上控制。

18.2.2 单机闭环

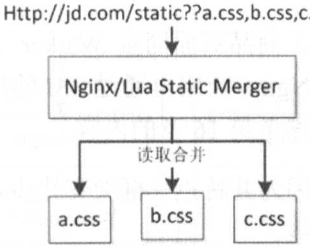
所谓单机闭环即所有想要的数据都能从本服务器中直接获取，在大多数时候无须通过网络去其他服务器获取。

如下图所示，主要有三种应用模式。



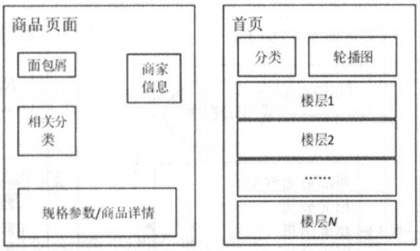
左边第一幅图的应用场景是 Nginx 应用谁也不依赖，比如，我们的 Cookie 白名单应用，其目的是不在白名单中的 Cookie 将被清理，防止大家随便将 Cookie 写到 jd.om 根下；大家访问 http://www.jd.com 时，会看到一个 http://ccc.jd.com/cookie_check 的请求用来清理 Cookie。对于这种应用非常简单，不需要依赖数据源，直接单应用闭环即可。

中间那幅的场景，是读取本机文件系统，如静态资源合并。比如，访问 http://item.jd.com/1856584.html，查看源码会发现（<link type="text/css" rel="stylesheet" href="//misc.360buyimg.com/jdf/1.0.0/unit/??ui-base/1.0.0/ui-base.css,shortcut/2.0.0/shortcut.css,global-header/1.0.0/global-header.css,myjd/2.0.0/myjd.css,nav/2.0.0/nav.css,shoppingcart/2.0.0/shoppingcart.css,global-footer/1.0.0/global-footer.css,service/1.0.0/service.css"/>）这种请求，即多个请求合并为一个发给服务器端，服务器端进行了文件资源的合并，如下图所示。



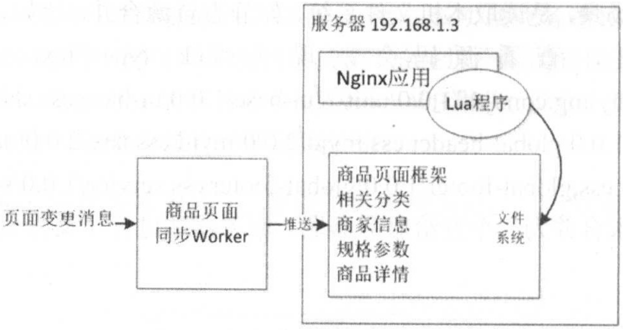
目前有成熟的 Nginx 模块（如 nginx-http-concat）进行静态资源合并。因为我们使用了 OpenResty，所以我们完全可以使用 Lua 编写程序实现该功能，比如，已经有人写了 nginx-lua-static-merger 来实现这个功能。

还有一些业务型应用场景如下图所示。



商品页面是由商品框架和其他维度的页面片段（面包屑、相关分类、商家信息、规格参数、商品详情）组成。或者首页是由首页框架和一些页面片段（分类、轮播图、楼层 1、楼层 *N*）组成。分维度是因为不同的维度是独立变化的。对于这种静态内容需要进行框架内容嵌入，Nginx 自带的 SSI（Server Side Include）可以很轻松地完成。也可以使用 Lua 程序更灵活地完成（读取框架、读取页面片段、合并输出）。

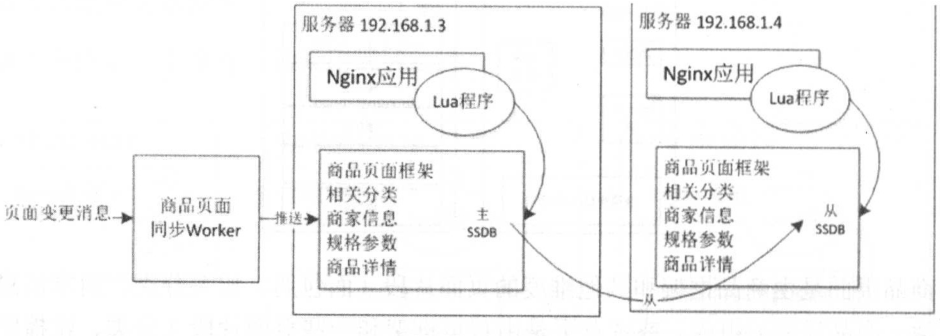
比如，商品页面的架构我们可以像下图这样实现。



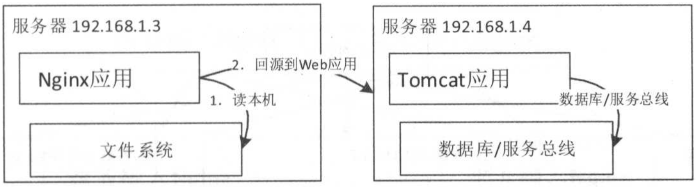
首先，接收到商品变更消息，商品页面同步 Worker 会根据消息维度生成相关的页面，然后推送到 Nginx 服务器。Nginx 应用再通过 SSI 输出。目前京东商品详情页没有再采用这种架构，具体架构可以参考第 16 章的内容。

对于首页的架构是类似的，因为其特点（框架变化少，楼层变化较频繁）和个性化的要求，楼层一般实现为异步加载。

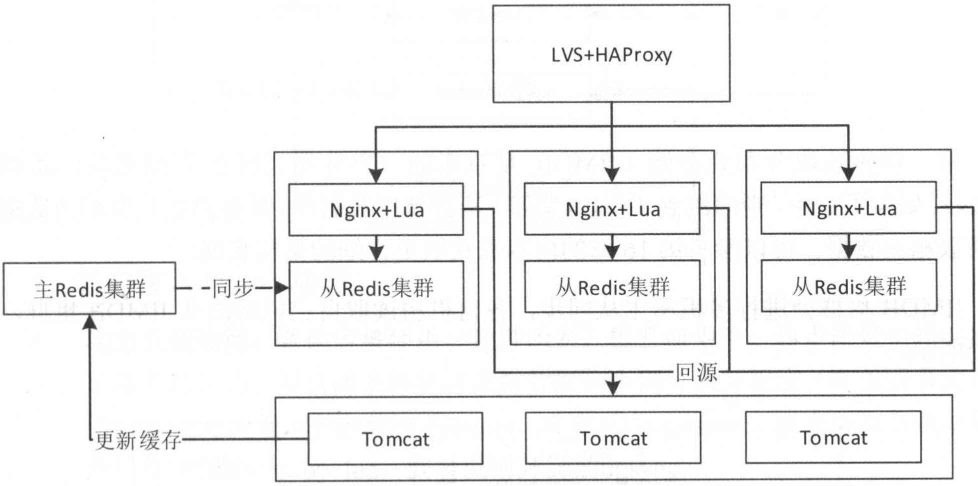
再来看右边那幅图。它与中间那幅图的不同之处是不再直接读取文件系统，而是读取本机的 Redis，或者 Redis 集群，或者如 SSDB 这种持久化存储，或者其他存储系统也是可以的，比如之前说的商品页面可以使用 SSDB 进行存储实现。文件系统存在一个很大的问题，即当有多台服务器时，需要 Worker 去写多台服务器，而这个过程可以使用 SSDB 的主从实现，如下图所示。



此处可以看到，不管是中间图还是右边图的架构，都需要 Worker 进行数据推送。假设本机数据丢了怎么办？因为有这个潜在问题，实际大部分应用不会是完全单机闭环的，而是会采用如下页图所示的架构。



或



即首先读本机，如果没数据，则会回源到相应的 Web 应用，从数据源拉取原始数据进行处理。这种架构的大部分场景本机都可以命中数据，只有很少一部分情况会回源到 Web 应用。

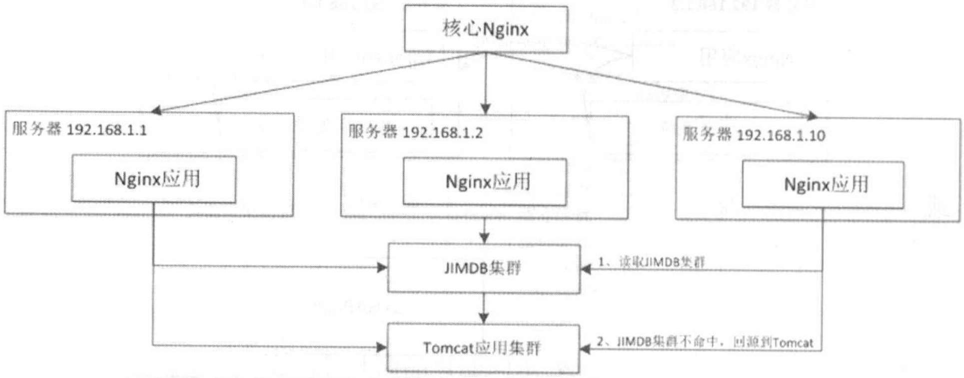
如京东的实时价格/动态服务就是采用类似架构。

18.2.3 分布式闭环

单机闭环会遇到如下两个主要问题：数据不一致问题（比如，没有采用主从架构导致不同服务器数据不一致）；存储瓶颈问题（磁盘或者内存遇到了天花板）。

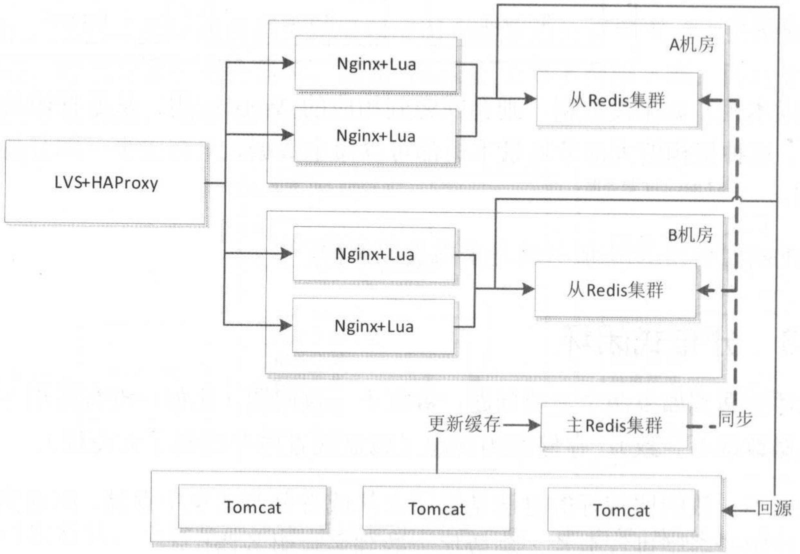
解决数据不一致的比较好的办法是采用主从或者分布式集中存储。而遇到存储瓶颈就需要进行按照业务键进行分片，将数据分散到多台服务器。

如采用如下页图所示的架构，按照尾号将内容分布到多台服务器中。



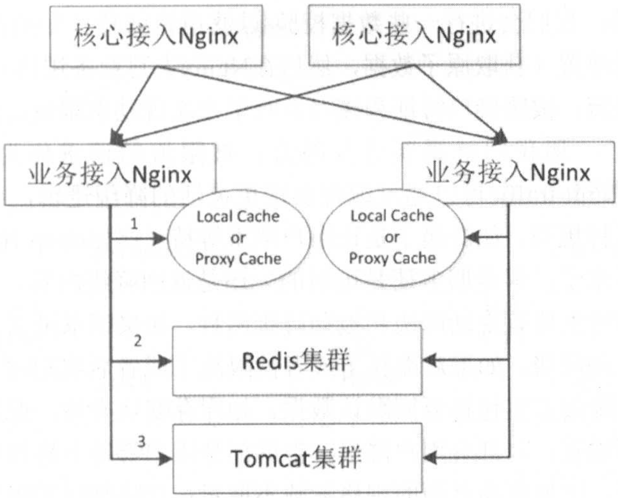
第一步先读取分布式存储（JIMDB 是京东的一个分布式缓存/存储系统，类似于 Redis）。如果不命中，则回源到 Tomcat 集群（其会调用数据库、服务总线获取相关数据）来获取相关数据。可以参考第 16 章的内容来获取更详细的架构实现。

JIMDB 集群会进行多机房主从同步，各自机房读取自己机房的从 JIMDB 集群，如下图所示。



18.2.4 接入网关

接入网关也可以叫做接入层，即接收到流量的入口，在入口我们可以进行如下页图所示的操作。



1. 核心接入 Nginx 功能

- **动态负载均衡：** 普通流量使用一致性哈希，提升命中率。热点流量走轮询减少单服务器压力。根据请求特征将流量分配到不同分组并限流（爬虫或者流量大的IP）。动态流量（动态增加upstream，或者减少upstream，或者动态负载均衡）可以使用balancer_by_lua，或者微博开源的upsync。
- **防DDoS攻击限流：** 可以将请求日志推送到实时计算集群，然后将需要限流的IP推送到核心Nginx进行限流。
- **非法请求过滤：** 比如，应该有Referer却没有，或者应该带着Cookie却没有Cookie。
- **请求聚合：** 比如请求的是http://c.3.cn/proxy?methods=a,b,c，核心接入Nginx会在服务端把Nginx的并发请求并把结果聚合然后一次性吐出。
- **请求头过滤：** 有些业务是不需要请求头的，因此，可以在往业务Nginx转发时把这些数据过滤掉。
- **缓存服务：** 使用Nginx Proxy Cache实现内容页面的缓存。

2. 业务 Nginx 功能

- **缓存：** 对于读服务会使用大量的缓存来提升性能，我们在设计时主要有如下缓存应用。首先，读取Nginx本地缓存Shared Dict或者Nginx Proxy Cache，如果有，则直接返回内容给用户。如果本地缓存不命中，则会读取分布式缓存如Redis，如果有，则直接返回。如果还是不命中，则回源到Tomcat应用读取DB或调用服务获取数据。另外，我们会按照维度进行数据缓存。

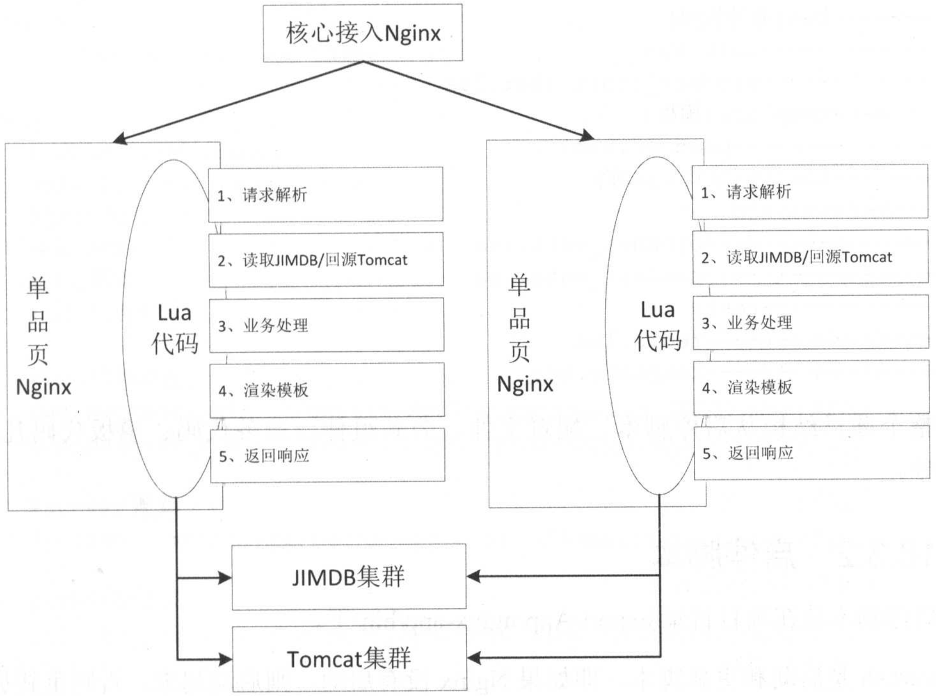
- **业务逻辑：** 我们会进行一些数据校验/过滤逻辑前置（如商品ID必须是数字）、业务逻辑前置（获取原子数据，然后在Nginx上写业务逻辑）。
- **细粒度限流：** 按照接口特征和接口吞吐量来实现动态限流，比如，后端服务快扛不住了，那我们就需要进行限流，被限流的请求作为降级请求处理。lua-resty-limit-traffic可以通过编程实现更灵活的降级逻辑，如根据用户、根据URL等各种规则，如降级了是让用户请求等待（比如sleep 100ms，这样用户请求就慢下来了，但是服务还是可用的）还是返回降级内容。
- **降级：** 降级主要有主动降级和被动降级两种。如果请求量太大扛不住了，那我们需要主动降级。如果后端挂了或者被限流了或者后端超时了，那我们需要被动降级。降级方案包括返回默认数据，如库存默认有货；返回静态页，如预先生成的静态页；对部分用户降级，告诉部分用户等待下再操作；直接降级，服务没数据，比如商品页面的规格参数不展示；只降级回源服务，即可以读取缓存的数据返回，实现部分可用，但是不会回源处理。
- **A/B测试和灰度发布：** 比如，要上一个新的接口，可以在业务Nginx通过Lua写复杂的业务规则实现不同的人看到不同的版本。
- **服务质量监控：** 我们可以记录请求响应时间、缓存响应时间、反向代理服务响应时间来详细了解到底哪块服务慢了。另外，记录非200状态码错误来了解服务的可用率。

京东的交易大 Nginx 节点、无线部门正在开发的无线 Nginx 网关和单品页统一服务都是接入网关的实践，而单品页统一服务架构可以参考第 17 章的内容。

18.2.5 Web 应用

此处所说的 Web 应用指的是页面模板渲染类型应用或者 API 服务类型应用。比如，京东列表页和商品详情页就是一个模板渲染类型的应用，核心业务逻辑都是使用 Lua 写的，部署到 Nginx 容器。目前核心业务代码行数有 5000 多行，模板页面有 2000 多行，涉及大量的计算逻辑，性能数据可以参考第 16 章的内容”。

如下页图所示，整体处理过程和普通 Web 应用没什么区别。首先，接收请求并进行解析；然后读取 JIMDB 集群数据，如果没有，则回源到 Tomcat 获取；最后进行业务逻辑处理，渲染模板，将响应内容返回给用户。



18.3 如何使用 OpenResty 开发 Web 应用

开发一个 Web 应用我们需要从项目搭建、功能开发、项目部署几个层面完成。

18.3.1 项目搭建

```
/export/App/nginx-app
 --------bin(脚本)
 -------------start.sh
 -------------stop.sh
 --------config(配置文件)
 -------------nginx.conf
 -------------domain
 -----------------nginx_product.conf
 -------------resources.properties
```

```
-------lua(业务代码)
------------init.lua
------------product_controller.lua
-------template(模板)
--------------prodoct.html
-------lualib(公共 Lua 库)
------------jd
----------------product_util.lua
----------------product_data.lua
------------resty
----------------redis.lua
----------------template.lua
```

整个项目结构从启停脚本、配置文件、公共组件、业务代码、模板代码几块进行划分。

18.3.2 启停脚本

启停脚本放在项目目录/export/App/nginx-app/bin/下。

start.sh 是启动和更新脚本，即如果 Nginx 没有启动，则启动起来，否则重新加载。

```
  sudo /export/servers/nginx/sbin/nginx  -t -c /export/App/nginx-app/
  config/nginx.conf
  sudo /export/servers/nginx/sbin/nginx  -c /export/App/nginx-app/
  config/nginx.conf
else
  sudo /export/servers/nginx/sbin/nginx -t
  sudo /export/servers/nginx/sbin/nginx  -s reload
end
```

stop.sh 是停止 Nginx 的脚本。

```
sudo /export/servers/nginx/sbin/nginx  -s quit
```

18.3.3 配置文件

配置文件放在/export/App/nginx-app/config 目录下，包括 nginx.conf 配置文件、Nginx 项目配置文件和资源配置文件。

18.3.4 Nginx.conf 配置文件

```
worker_processes  1;
```

```
events {
    worker_connections 1024;
}
http {
   include mime.types;
   default_type  text/html;
   #gzip 相关
   #超时时间
   #日志格式
   #反向代理配置

   #Lua 依赖路径
   lua_package_path  "/export/App/nginx-app/lualib/?.lua;;";
   lua_package_cpath  "/export/App/nginx-app/lualib/?.so;;";

   #server 配置
   include /export/App/nginx-app/config/domains/*;

   #初始化脚本
   init_by_lua_file "/export/App/nginx-app/lua/init.lua";
}
```

对于 nginx.conf 会进行一些通用的配置，如工作进程数、超时时间、压缩、日志格式、反向代理等相关配置。另外，需要指定如下配置。

- **lua_package_path、lua_package_cpath**：指定我们依赖的通用Lua库从哪里加载。
- **include /export/App/nginx-app/config/domains/***：用于加载Server相关配置，此处通过*可以在一个Nginx下指定多个Server配置。
- **init_by_lua_file "/export/App/nginx-app/lua/init.lua"**：执行项目的一些初始化配置，比如加载配置文件。

18.3.5 Nginx 项目配置文件

/export/App/nginx-app/config/domains/nginx_product.conf 用于配置当前 Web 应用的一些 Server 相关配置。

```
#upstream
upstream item_http_upstream {
    server 192.168.1.1 max_fails=2 fail_timeout=30s weight=5;
    server 192.168.1.2 max_fails=2 fail_timeout=30s weight=5;
}
```

```
#缓存
lua_shared_dict item_local_shop_cache 600m;
server {
    listen              80;
    server_name         item.jd.com item.jd.hk;
    #模板文件从哪加载
   set $template_root "/export/App/nginx-app/template ";
    #URL 映射
      location ~* "^/product/(\d+)\.html$" {
         rewrite /product/(.*)   http://item.jd.com/$1 permanent;
      }
 location ~* "^/(\d{6,12})\.html$" {
         default_type text/html;
         charset gbk;
         lua_code_cache on;
         content_by_lua_file
"/export/App/nginx-app/lua/product_controller.lua";
      }
}
```

我们需要指定如 upstream、共享字典配置、Server 配置、模板文件从哪加载、URL 映射，比如我们访问 http://item.jd.com/1856584.html 将交给/export/App/nginx-app/lua/product_controller.lua 来处理，也就是说我们项目的入口就有了。

资源配置文件 resources.properties 包含了我们的一些比如开关的配置、缓存服务器地址的配置等。

18.3.6 业务代码

/export/App/nginx-app/lua/目录里存放了我们的 Lua 业务代码，init.lua 用于读取如 resources.properties 来进行一些项目初始化。product_controller.lua 可以看成 Java Web 中的 Servlet，用来接收、处理、响应用户请求。

18.3.7 模板

模板文件放在/export/App/nginx-app/template/目录下，使用相应的模板引擎进行编写页面模板，然后渲染输出。

18.3.8 公共 Lua 库

存放了一些如 Redis、Template 等相关的公共 Lua 库，还有一些我们项目中通用的

工具库如 product_util.lua。

至此一个简单的项目结构就介绍完了，对于开发一个项目来说，还会涉及分模块等工作，不过，对于我们这种 Lua 应用来说，建议不要过度抽象，尽量小巧即可。

18.3.9 功能开发

接下来，就需要使用相应的 API 来实现我们的业务了，比如 product_controller.lua。

```
--加载 Lua 模块库
local template = require("resty.template")
--1. 获取请求参数中的商品 ID
local skuId = ngx.req.get_uri_args()["skuId"];
--2. 调用相应的服务获取数据
local data = api.getData(skuId)
--3. 渲染模板
local func = template.compile("product.html")
local content = func(data)
--4. 通过 ngx API 输出内容
ngx.say(content)
```

开发完成后将项目部署到测试环境，执行 start.sh 启动 Nginx，然后进行测试。详细的开发过程和 API 的使用，请参考《跟我学 OpenResty（Nginx+Lua）开发》，此处不做具体编码实现。若要参考源码请访问 https://github.com/zhangkaitao/openresty-helloworld。

18.4 基于 OpenResty 的常用功能总结

到此我们对于 Nginx 开发已经有了一个整体认识，将 Nginx 黏合 Lua 来开发应用可以说是一把锋利的瑞士军刀，可以帮我们很容易地解决很多问题，可以开发 Web 应用、接入网关、API 网关、消息推送、日志采集等应用，不过，个人认为适合开发业务逻辑单一、核心代码行数较少的应用，不适合业务逻辑复杂、功能繁多的业务型或者企业级应用。最后我们总结一下基于 Nginx+Lua 的常用架构模式中的一些常见实践和场景，包括：动态负载均衡、防火墙（DDoS、IP/URL/UserAgent/Referer 黑名单、防盗链等）、限流、降级、A/B 测试和灰度发布、多级缓存模式、服务器端请求聚合、服务质量监控。

18.5 一些问题

- 在开发Nginx应用时，使用UTF-8编码可以减少很多麻烦。
- GBK转码解码时，应使用GB18030，否则一些特殊字符会出现乱码。
- cjson库对于像\uab1这种错误的unicode转码会失败，可以使用纯Lua编写的dkjson。
- 社区版Nginx不支持upstream的域名动态解析，可以考虑proxy_pass（http://p.3.local/prices/mgetsis_argsargs），然后配合resolver来实现。或者在Lua中进行HTTP调用。如果DNS遇到性能瓶颈，则可以考虑在本机部署如dnsmasq来缓存，或者考虑使用balancer_by_lua功能实现动态upstream。
- 为响应添加处理服务器IP的响应头，方便定位问题。
- 根据业务设置合理的超时时间。
- 运行CDN的业务，当发生错误时，不要给返回的500/503/302/301等非正常响应设置缓存。

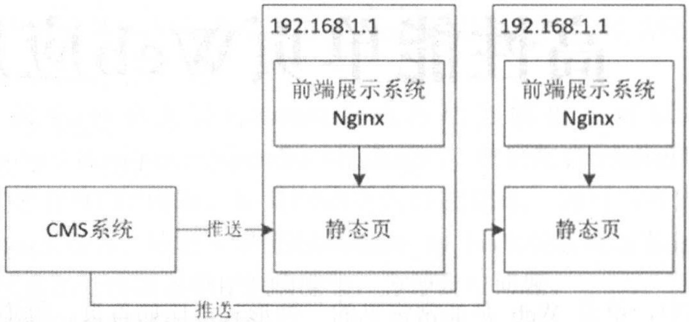
19 应用数据静态化架构高性能单页Web应用

在电商网站中，单页 Web 是非常常见的一种形式，比如首页、频道页、广告页等都属于单页应用。而这种页面是由模板+数据组成的。传统的构建方式一般通过静态化实现，但这种方式的灵活性并不是很好，比如，页面模板部分变更了需要重新全部生成。因此，最好能有一种实现方式是可以实时动态渲染的，以支持模板的多变性。另外也要考虑好如下几个问题。

- 动态化模板渲染支持。
- 数据和模板的多版本化：生产版本、灰度版本和预发布版本。
- 版本回滚问题，假设当前发布的生产版本出问题了，如何快速回滚到上一个版本。
- 异常问题，假设渲染模板时，遇到了异常情况（比如获取Redis出问题了），该如何处理。
- 灰度发布问题，比如切20%量给灰度版本。
- 预发布问题，目的是在正式环境测试数据和模板的正确性。

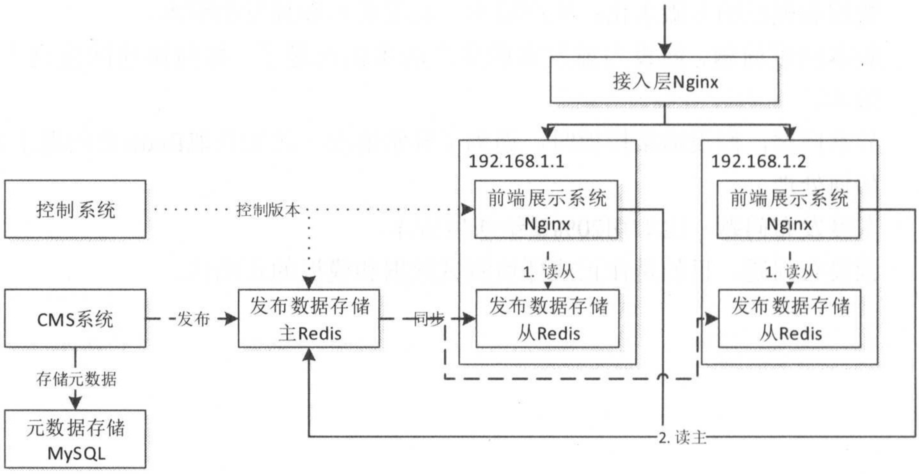
19.1 整体架构

静态化页面的方案如下图所示。



直接将生成的静态页推送到相关服务器上即可。使用这种方式要考虑文件操作的原子化问题（即从老版本切换到新版本如何做到文件操作原子化）。

而动态化方案的整体架构如下图所示，分为三大系统：CMS 系统、控制系统和前端展示系统。



19.1.1 CMS 系统

在 CMS 系统可以配置页面的模板和数据。

- 模板动态在CMS系统中维护，即模板不是一个静态文件，而是存储在CMS中的一条数据，最终发布到“发布数据存储Redis”中，前端展示系统从Redis中获取该模板进行渲染，从而前端展示系统更换了模板也不需要重启，纯动态维护模板数据。
- 原始数据存储到“元数据存储MySQL”中即可，比如，频道页一般需要前端访问的URL、分类、轮播图、商品楼层数据等。这些数据按照相应的维度存储在CMS系统中。
- 提供发布到“发布数据存储Redis”的控制，将CMS系统中的原始数据和模板数据组装成聚合数据（JSON存储）同步到“发布数据存储Redis”，以便前端展示系统获取进行展示。此处提供三个发布按钮：正式版本、灰度版本和预发布版本。

目前存在如下几个问题。

- 用户如访问http://channel.jd.com/fashion.html，怎么定位到对应的聚合数据呢？我们可以在CMS元数据中定义URL作为key，如果没有URL，则使用ID作为key，或者自动生成一个URL。
- 多版本如何存储呢？使用Redis的Hash结构存储即可，key为URL（比如http://channel.jd.com/fashion.html），字段按照维度存储：正式版本使用当前时间戳存储（这样前端系统可以根据时间戳排序，然后获取最新版本）、预发布版本使用“predeploy”作为字段，灰度版本使用“abVersion”作为字段即可，这样就区分开了多版本。
- 灰度版本如何控制呢？可以通过控制系统的开关来控制如何灰度。
- 如何访问预发布版本呢？比如，在URL参数中常加上predeploy=true，另外，可以限定只有内网可以访问或者访问时加上访问密码，比如pwd=absdfedwqdqw。
- 如何处理模板变更的历史数据校验问题？比如模板变更了，但是，使用历史数据渲染该模板会出现问题，即模板要兼容历史数据的。此处的方案不存在这个问题，因为每次存储时是当时的模板快照，即数据快照和模板快照推送到“发布数据存储Redis”中。

19.1.2 前端展示系统

- 获取当前URL，使用URL作为key首先从本机“发布数据存储Redis”获取数据。

- 如果没有数据或者异常，则从主“发布数据存储Redis”获取。
- 如果主“发布数据存储Redis”也发生了异常，那么会直接调用CMS系统暴露的API，直接从元数据存储MySQL中获取数据进行处理。

展示系统的伪代码

```
--1. 加载 Lua 模块库
local template = require("resty.template")
template.load = function(s) return s end
--2. 动态获取模板
local myTemplate = "<html>{* title *}</html>"
--3. 动态获取数据
local data = {title = "iphone6s"}
--4. 渲染模板
local func = template.compile(myTemplate)
local content = func(data)
--5. 通过 ngx API 输出内容
ngx.say(content)
```

即模板和数据都是动态获取的，然后使用动态获取的模板和数据进行渲染。

假设最新版本的模板或数据有问题怎么办？可以从流程上避免这个问题：首先进行预发布版本发布，测试人员验证没问题后，接着发布灰度版本；在灰度时自动去掉 CDN 功能（即不设置页面的缓存时间），发布验证没问题后，发布正式版本即可；正式版本发布的 5 分钟内是不设置页面缓存的，这样就可以防止发版时遇到问题，但是，问题版本已经在 CDN 上给全部用户造成问题了。当然这个流程很麻烦，可以按照自己的场景进行简化。

19.1.3 控制系统

控制系统用于版本降级和灰度发布，当然可以把这个功能放在 CMS 系统中实现。

- **版本降级**：假设当前线上的版本遇到问题，想要快速切换回上一个版本，可以使用控制系统实现，选中其中一个历史版本，然后通知给前端展示系统即可，使用URL和当前版本的字段即可，这样前端展示系统就可以自动切换到选中的那个版本。当问题修复后，再删除该降级配置，即切换回最新版本。
- **灰度发布**：在控制系统中控制哪些URL需要灰度发布，以及灰度发布的比例，同版本降级类似将相关的数据推送到前端展示系统，当不想灰度发布时，删除相关数据即可。

19.2 数据和模板动态化

我们将数据和模板都进行动态化存储，这样可以在 CMS 进行数据和模板的变更。实现前端和后端开发人员的分离。前端开发人员进行 CMS 数据配置和模板开发，而后端开发人员只进行系统维护。另外，因为模板的动态化存储，每次发布新的模板不需要重启前端展示系统，后端开发人员更好地得到了解放。

模板和数据可以是一对多的关系，即一个模板可以被多个数据使用。当模板发生变更后，我们可以批量推送模板关联的数据。首先，进行预发布版本的发布，测试人员进行验证，验证没问题即可发布正式版本。

19.3 多版本机制

我们将数据和模板分为多版本后可以实现如下几点。

- **预发布版本**：更容易让测试人员在实际环境中进行验证。
- **灰度版本**：只需要简单的开关控制，就可以进行A/B测试。
- **正式版本**：存储多个历史正式版本，假设最新的正式版本出现问题，可以非常快速地切换回之前的版本。

19.4 异常问题

常见的几种异常如下。

- 本机从“发布数据存储Redis”和主“发布数据存储Redis”都不能用了，那么，可以直接调用CMS系统暴露的HTTP服务，直接从元数据存储MySQL获取数据。
- 数据和模板获取到了，但是渲染模板出错了，比如遇到500、503。解决方案是使用上一个版本的数据进行渲染。
- 数据和模板都没问题，但是因为一些疏忽，渲染出来的页面错乱了，或者有些区域出现了空白。对于这种问题没有很好的解决方案。可以根据自己的场景定义异常扫描库，扫描到当前版本有异常就发警告给相关人员，并自动降级到上一个版本。

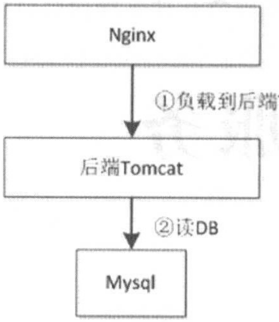
20 使用OpenResty开发Web服务

本文所说的 HTTP 服务主要指如访问京东网站时我们看到的热门搜索、用户登录、实时价格、实时库存、服务支持、广告语等这种非 Web 页面，这些是在如商品详情页中异步加载的相关数据，它们有个特点——访问量巨大、逻辑比较单一。但是，实时库存逻辑其实是非常复杂的。在京东这些服务每天有几亿、十几亿的访问量。比如，实时库存服务曾经在没有任何 IP 限流、DDoS 防御的情况被刷到 600 多万/分钟的访问量，仍然能轻松应对。支撑如此大的访问量，就需要考虑设计良好的架构，并确保很容易实现水平扩展。

20.1 架构

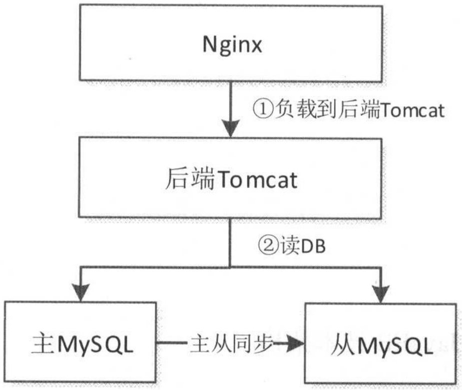
此处介绍一下笔者曾使用过的 OpenResty+JavaEE 技术架构。

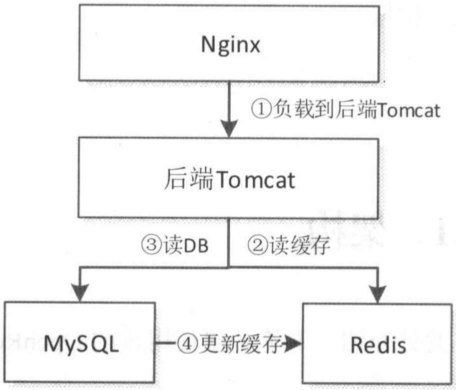
20.2 单 DB 架构



早期架构可能就是 Nginx 直接 upstream 请求到后端 Tomcat，扩容时基本是增加新的 Tomcat 实例，然后通过 Nginx 负载均衡 upstream 过去，此时数据库还不是瓶颈。当访问量到一定级别，数据库的压力就上来了，此时单纯地靠单个数据库可能扛不住，可以通过数据库的读写分离或加缓存来应对。

20.2.1 DB+Cache/数据库读写分离架构

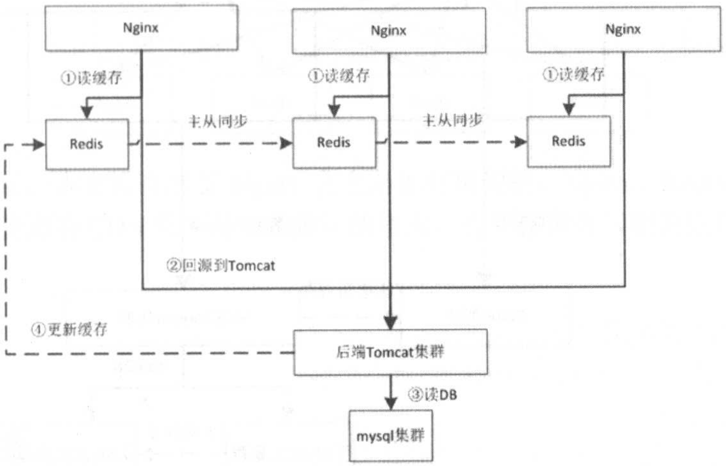




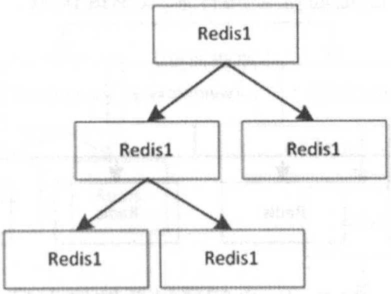
此时通过使用如数据库读写分离或者 Redis 这种缓存来支撑更大的访问量。使用缓存这种架构会遇到的问题，包括缓存与数据库数据不同步造成数据不一致（一般设置过

期时间），或者 Redis 不可用时直接命中数据库导致数据库压力过大。可以考虑 Redis 的主从或者用一致性哈希算法做分片的 Redis 集群。使用缓存这种架构，要求应用对数据一致性的要求不是很高。比如，像下订单这种要落地的数据，则不适合用 Redis 存储，但是，订单的读取可以使用缓存。

20.2.2 OpenResty+Local Redis+MySQL 集群架构



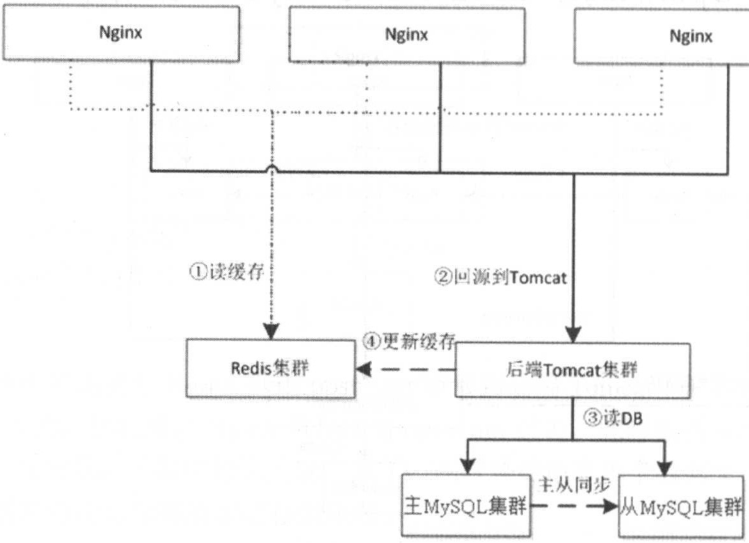
如上图所示，OpenResty 首先通过 Lua 读取本机 Redis 缓存，如果不命中，则回源到后端 Tomcat 集群。后端 Tomcat 集群再读取 MySQL 数据库。Redis 都是安装到和 OpenResty 同一台服务器上，OpenResty 直接读本机可以减少网络延时。Redis 通过主从方式同步数据，Redis 主从一般采用树的方式实现。



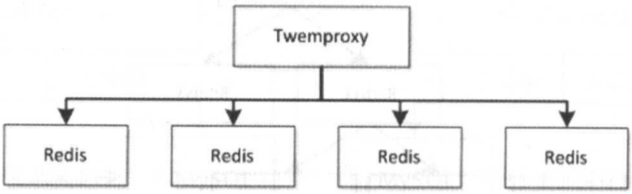
如上图所示，在叶子节点可以做AOF持久化，保证在主Redis不可用时能进行恢复。如对 Redis 很依赖，可以考虑多主 Redis 架构，而不是单主，来防止单主不可用时数据

的不一致和击穿到后端 Tomcat 集群。这种架构的缺点就是要求 Redis 实例数据量较小，如果单机内存不足，那么也可以通过如尾号为 1 的在 A 服务器、尾号为 2 的在 B 服务器这种方式实现。缺点也很明显，运维复杂、扩展性差。

20.2.3 OpenResty+Redis 集群+MySQL 集群架构



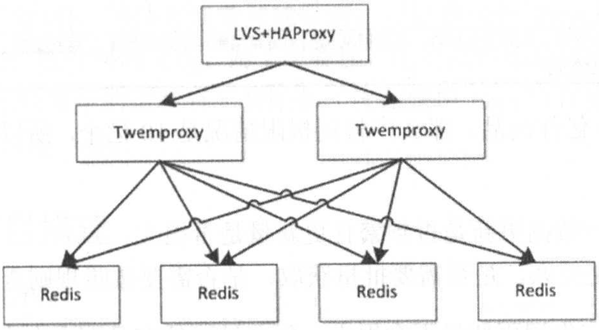
如上图所示，与之前架构不同的是，此时我们使用一致性哈希算法实现 Redis 集群，而不是读本机 Redis，保证其中一台不可用时，只有很少的数据会丢失，防止击穿到数据库。Redis 集群分片可以使用 Twemproxy。如果 Tomcat 实例很多的话，就要考虑 Redis 和 MySQL 链接数问题，因为大部分 Redis/MySQL 客户端都是通过连接池实现，此时的链接数会成为瓶颈。一般方法是通过中间件来减少链接数。



如上图所示，Twemproxy 与 Redis 之间通过单链接交互，并通过 Twemproxy 实现分片逻辑。这样我们可以水平扩展更多的 Twemproxy 来增加链接数。

此时的问题就是 Twemproxy 实例众多，应用维护、配置困难，需要在这之上做负

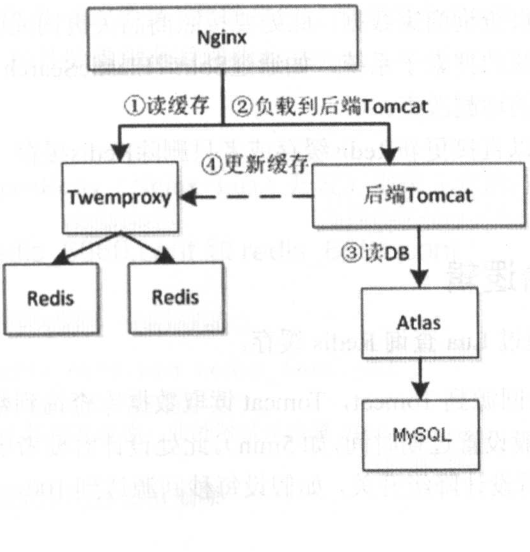
载均衡，比如，通过 LVS/HaProxy 实现 VIP（虚拟 IP），可以做到切换对应用透明、故障自动转移。还可以通过实现内网 DNS 来做其负载均衡。



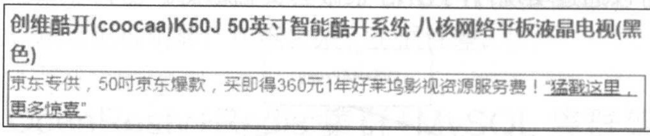
如上图所示，本文没有涉及 Nginx 之上是如何架构的，Nginx、Redis、MySQL 等的负载均衡、资源的 CDN 化不是本章关注的重点，有兴趣请查阅相关资料。

20.3 实现

接下来，我们来搭建一下第四种架构，如下图所示。



以获取如京东商品页广告词为例，如下图所示。



假设京东有 10 亿件商品，那么广告词极限情况是 10 亿个，所以在设计时就要考虑以下内容。

- 数据量——数据更新是否频繁且更新量是否很大。
- 是K-V还是关系，是否需要批量获取，是否需要按照规则查询。

而对于本例，广告词更新量不会很大，每分钟可能在几万个左右。而且是 K-V 的，其实适合使用关系存储。因为广告词是商家在维护，因此后台查询需要知道这些商品是哪个商家的。而前台是不关心商家的，是 KV 存储，所以前台显示可以放进如 Redis 中，即存在两种设计。

- 所有数据存储到MySQL，然后热点数据加载到Redis。
- 关系存储到MySQL，而数据存储到如SSDB这种持久化KV存储中。

基本数据结构包括商品 ID、广告词、所属商家、开始时间、结束时间、是否有效。

20.3.1 后台逻辑

- 商家登录后台。
- 按照商家分页查询商家数据，此处要按照商品关键词或商品类目查询的话，需要走商品系统的搜索子系统，如通过Solr或ElasticSearch实现搜索子系统。
- 进行广告词的增删改查。
- 增删改时可以直接更新Redis缓存或者只删除Redis缓存（第一次前台查询时写入缓存）。

20.3.2 前台逻辑

1．首先 Nginx 通过 Lua 查询 Redis 缓存。

2．查询不到的话回源到 Tomcat，Tomcat 读取数据库查询到数据，然后把最新的数据异步写入 Redis(一般设置过期时间，如 5min)。此处设计时要考虑 Tomcat 读取 MySQL 的极限值是多少，然后设计降级开关，如假设每秒回源达到 100，则直接不查询 MySQL

而返回空的广告词来防止 Tomcat 应用雪崩。

为了简单，我们不进行后台的设计实现，只做前端的设计实现，此时数据结构我们简化为[商品 ID、广告词]。另外有读者可能看到，可以直接把 Tomcat 部分干掉，通过 Lua 直接读取 MySQL 进行回源实现。为了完整性，此处我们还是做回源到 Tomcat 的设计，因为如果逻辑比较复杂的话会有一些限制（比如使用 Java 特有协议的 RPC），还是通过 Java 去实现更方便一些。

20.3.3 项目搭建

项目部署目录结构。

```
/usr/chapter6
  redis_6660.conf
  redis_6661.conf
  nginx_chapter6.conf
  nutcracker.yml
  nutcracker.init
  webapp
    WEB-INF
      lib
      classes
      web.xml
```

20.3.4 Redis+Twemproxy 配置

根据实际情况来决定 Redis 大小，此处我们已经有两个 Redis 实例（6660、6661），在 Twemproxy 上通过一致性哈希做分片逻辑。

1. 安装

请参考《跟我学 OpenResty（Nginx+Lua）开发》中第 3 章的内容。

2. Redis 配置 redis_6660.conf 和 redis_6661.conf

```
#分别为 6660 6661
port 6660
#进程 ID 分别改为 redis_6660.pid redis_6661.pid
pidfile "/var/run/redis_6660.pid"
#设置内存大小，根据实际情况设置，此处测试仅设置 20MB
maxmemory 20mb
#内存不足时，按照过期时间进行 LRU 删除
```

```
maxmemory-policy volatile-lru
#Redis 的过期算法不是精确的而是通过采样来算的，默认采样为 3 个，此处我们改成 10
maxmemory-samples 10
#不进行 RDB 持久化
save ""
#不进行 AOF 持久化
appendonly no
```

将如上配置放到 redis_6660.conf 和 redis_6661.conf 配置文件最后即可，后边的配置会覆盖前边的。

3. Twemproxy 配置 nutcracker.yml

```
server1:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  servers:
   - 127.0.0.1:6660:1 server1
   - 127.0.0.1:6661:1 server2
```

复制 nutcracker.init 到/usr/chapter6 下，并修改配置文件为/usr/chapter6/nutcracker.yml。

4. 启动

```
nohup /usr/servers/redis-2.8.19/redis-server /usr/chapter6/redis_6660. conf &
nohup /usr/servers/redis-2.8.19/redis-server /usr/chapter6/redis_6661. conf &
/usr/chapter6/nutcracker.init start
ps -aux | grep -e redis  -e nutcracker
```

20.3.5 MySQL+Atlas 配置

Atlas 类似于 Twemproxy，是 Qihoo 360 基于 MySQL Proxy 开发的一个 MySQL 中间件，据称每天承载读写请求数达几十亿，可以实现分库（sharding 版本）、分表、读写分离、数据库连接池等功能，缺点是没有实现跨库分表（分库）功能，需要在客户端使用分库逻辑。另一个选择是使用如阿里的 TDDL，它是在客户端实现分库的。到底选择是在客户端还是在中间件根据实际情况来选择。

此处我们不做 MySQL 的主从复制（读写分离），只做分库分表实现。

1. MySQL初始化

为了测试我们此处分两个表（N=0/1）。

```
CREATE DATABASE chapter6 DEFAULT CHARACTER SET utf8;
use chapter6;
CREATE TABLE  chapter6.ad_N(
     sku_id BIGINT,
     content VARCHAR(4000)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8;
```

2. Atlas安装

```
cd /usr/servers/
wget https://github.com/Qihoo360/Atlas/archive/2.2.1.tar.gz  -O  Atlas-2.2.
1.tar .gz
tar -xvf Atlas-2.2.1.tar.gz
cd Atlas-2.2.1/
#Atlas依赖mysql_config，如果没有，则可以通过如下方式安装
apt-get install libmysqlclient-dev
#安装Lua依赖
wget http://www.lua.org/ftp/lua-5.1.5.tar.gz
tar -xvf lua-5.1.5.tar.gz
cd lua-5.1.5/
make linux && make install
#安装glib依赖
apt-get install libglib2.0-dev
#安装libevent依赖
apt-get install libevent
#安装flex依赖
apt-get install flex
#安装jemalloc依赖
apt-get install libjemalloc-dev
#安装OpenSSL依赖
apt-get install openssl
apt-get install libssl-dev
apt-get install libssl0.9.8

./configure --with-mysql=/usr/bin/mysql_config
./bootstrap.sh
make && make install
```

3. Atlas配置

```
vim /usr/local/mysql-proxy/conf/chapter6.cnf
```

```
[mysql-proxy]
#Atlas 代理的主库，多个之间逗号分隔
proxy-backend-addresses = 127.0.0.1:3306
#Atlas 代理的从库，多个之间逗号分隔，格式 ip:port@weight，权重默认为 1
#proxy-read-only-backend-addresses = 127.0.0.1:3306,127.0.0.1:3306
#用户名/密码，密码使用/usr/servers/Atlas-2.2.1/script/encrypt 123456 加密
pwds = root:/iZxz+0GRoA=
#后端进程运行
daemon = true
#开启 monitor 进程，当 worker 进程挂了自动重启
keepalive = true
#工作线程数，对 Atlas 的性能有很大影响，可根据情况适当设置
event-threads = 64
#日志级别
log-level = message
#日志存放的路径
log-path = /usr/chapter6/
#实例名称，用于同一台机器上多个 Atlas 实例间的区分
instance = test
#监听的 ip 和 port
proxy-address = 0.0.0.0:1112
#监听的管理接口的 ip 和 port
admin-address = 0.0.0.0:1113
#管理接口的用户名
admin-username = admin
#管理接口的密码
admin-password = 123456
#分表逻辑
tables = chapter6.ad.sku_id.2
#默认字符集
charset = utf8
```

因为本例没有做读写分离，所以读库 proxy-read-only-backend-addresses 没有配置。分表逻辑为：数据库名.表名.分表键.表的个数，分表的表名格式是 table_N，N 从 0 开始。

4. Atlas 启动/重启/停止

```
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 start
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 restart
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 stop
```

如上命令会自动到/usr/local/mysql-proxy/conf 目录下查找 chapter6.cnf 配置文件。

5. Atlas 管理

通过如下命令进入管理接口。

```
mysql -h127.0.0.1 -P1113  -uadmin -p123456
```

通过执行 SELECT * FROM help 查看帮助。还可以通过一些 SQL 进行服务器的动态添加/移除。

6. Atlas 客户端

通过如下命令进入客户端接口。

```
mysql -h127.0.0.1 -P1112  -uroot -p123456
use chapter6;
insert into ad values(1 '测试1);
insert into ad values(2, '测试2');
insert into ad values(3 '测试3);
select * from ad where sku_id=1;
select * from ad where sku_id=2;
#通过如下 SQL 可以看到实际的分表结果
select * from ad_0;
select * from ad_1;
```

此时无法执行 select * from ad，需要使用如“select * from ad where sku_id=1”这种 SQL 进行查询。即需要带上 sku_id 且必须是相等比较。如果是范围或模糊查询，那么是不可以的。如果想全部查询，则只能遍历所有表进行查询，即在客户端做查询-聚合。

此处实际的分表逻辑是按照商家进行分表，而不是按照商品编号，因为我们后台查询时是按照商家维度，此处是为了测试才使用商品编号的。

至此基本的 Atlas 就介绍完了，更多内容请参考如下资料。

MySQL 主从复制：http://369369.blog.51cto.com/319630/790921/

MySQL 中间件介绍：http://www.guokr.com/blog/475765/

Atlas 使用：http://www.0550go.com/database/mysql/mysql-atlas.html

Atlas 文档：https://github.com/Qihoo360/Atlas/blob/master/README_ZH.md

20.3.6 Java+Tomcat 安装

1．Java 安装

```
cd /usr/servers/
#首先到如下网站下载 JDK
#http://www.oracle.com/technetwork/cn/java/javase/downloads/jdk7-downloads-1880260.html
#本文下载的是 jdk-7u75-linux-x64.tar.gz。
tar -xvf jdk-7u75-linux-x64.tar.gz
vim ~/.bashrc
```

在文件最后添加如下环境变量。

```
export JAVA_HOME=/usr/servers/jdk1.7.0_75/
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH
export CLASSPATH=$CLASSPATH:.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib

#使环境变量生效
source ~/.bashrc
```

2．Tomcat 安装

```
cd /usr/servers/
wget
http://ftp.cuhk.edu.hk/pub/packages/apache.org/tomcat/tomcat-7/v7.0.59/bin/apache-tomcat-7.0.59.tar.gz
tar -xvf apache-tomcat-7.0.59.tar.gz
cd apache-tomcat-7.0.59/
#启动
/usr/servers/apache-tomcat-7.0.59/bin/startup.sh
#停止
/usr/servers/apache-tomcat-7.0.59/bin/shutdown.sh
#删除 Tomcat 默认的 webapp
rm -r apache-tomcat-7.0.59/webapps/*
#通过 Catalina 目录发布 Web 应用
cd apache-tomcat-7.0.59/conf/Catalina/localhost/
vim ROOT.xml
```

3．ROOT.xml

```
<!-- 访问路径是根，Web 应用所属目录为/usr/chapter6/webapp -->
<Context path="" docBase="/usr/chapter6/webapp"></Context>
#创建一个静态文件随便添加点内容
vim /usr/chapter6/webapp/index.html
#启动
```

```
/usr/servers/apache-tomcat-7.0.59/bin/startup.sh
```

访问 http://192.168.1.2:8080/index.html，如能处理内容说明配置成功。

```
#变更目录结构
cd /usr/servers/
mv apache-tomcat-7.0.59 tomcat-server1
#此处我们创建两个 Tomcat 实例
cp -r tomcat-server1 tomcat-server2
vim tomcat-server2/conf/server.xml
#如下端口进行变更
8080--->8090
8005--->8006
```

启动两个 Tomcat。

```
/usr/servers/tomcat-server1/bin/startup.sh
/usr/servers/tomcat-server2/bin/startup.sh
```

分别访问如下两个地址，如果能正常访问，则说明配置正常。

http://192.168.1.2:8080/index.html

http://192.168.1.2:8090/index.html

如上步骤使我们在一个服务器上能启动两个 Tomcat 实例，这样的好处是我们可以做本机的 Tomcat 负载均衡，假设一个 Tomcat 重启时另一个是可以工作的，从而不至于不给用户返回响应。

20.3.7 Java+Tomcat 逻辑开发

1. 搭建项目

我们使用 Maven 搭建 Web 项目，Maven 知识请自行学习。

2. 项目依赖

本文将最小化依赖，即仅依赖我们需要的 Servlet、MySQL、Druid、Jedis。

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
```

```
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.27</version>
    </dependency>
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.0.5</version>
    </dependency>
    <dependency>
      <groupId>redis.clients</groupId>
      <artifactId>jedis</artifactId>
      <version>2.5.2</version>
    </dependency>
  </dependencies>
```

3. 核心代码

编辑 com.github.zhangkaitao.chapter6.servlet.AdServlet 代码。

```
public class AdServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String idStr = req.getParameter("id");
        Long id = Long.valueOf(idStr);
        //1. 读取 MySQL 获取数据
        String content = null;
        try {
            content = queryDB(id);
        } catch (Exception e) {
            e.printStackTrace();
            resp.setStatus(
                    HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
              return;
        }
          if(content != null) {
            //2.1 如果获取到，则异步写 Redis
            asyncSetToRedis(idStr, content);
            //2.2 如果获取到，则把响应内容返回
            resp.setCharacterEncoding("UTF-8");
            resp.getWriter().write(content);
        } else {
```

```
            //2.3 如果获取不到，则返回 404 状态码
            resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
        }
    }

    private DruidDataSource datasource = null;
    private JedisPool jedisPool = null;

    {
        datasource = new DruidDataSource();
        datasource.setUrl(
                "jdbc:mysql://127.0.0.1:1112/chapter6?useUnicode=true&c
haracterEncoding=utf-8&autoReconnect=true");
        datasource.setUsername("root");
        datasource.setPassword("123456");
        datasource.setMaxActive(100);

        GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();
        poolConfig.setMaxTotal(100);
        jedisPool = new JedisPool(poolConfig, "127.0.0.1", 1111);
    }

    private String queryDB(Long id) throws Exception {
        Connection conn = null;
        try {
            conn = datasource.getConnection();
            String sql = "select content from ad where sku_id = ?";
            PreparedStatement psst = conn.prepareStatement(sql);
            psst.setLong(1, id);
            ResultSet rs = psst.executeQuery();
            String content = null;
            if(rs.next()) {
                content = rs.getString("content");
            }
            rs.close();
            psst.close();
            return content;
        } catch (Exception e) {
            throw e;
        } finally {
            if(conn != null) {
                conn.close();
            }
        }
```

```
    }

    private ExecutorService executorService = Executors.newFixedThreadPool(10);
    private void asyncSetToRedis(final String id, final String content) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                Jedis jedis = null;
                try {
                    jedis = jedisPool.getResource();
                    jedis.setex(id, 5 * 60, content);//5 分钟
                } catch (Exception e) {
                    e.printStackTrace();
                    jedisPool.returnBrokenResource(jedis);
                } finally {
                    jedisPool.returnResource(jedis);
                }
            }
        });
    }
}
```

整个逻辑比较简单，此处更新缓存一般使用异步方式去实现，这样不会阻塞主线程。另外，此处可以考虑使用 Servlet 异步化来提示吞吐量。

4．web.xml 配置

```
<servlet>
    <servlet-name>adServlet</servlet-name>

<servlet-class>com.github.zhangkaitao.chapter6.servlet.AdServlet</servlet
-class>
</servlet>
<servlet-mapping>
    <servlet-name>adServlet</servlet-name>
    <url-pattern>/ad</url-pattern>
</servlet-mapping>
```

5．打 WAR 包

```
cd D:\workspace\chapter6
mvn clean package
```

此处使用 maven 命令打包。比如，本例将得到 chapter6.war，然后将其上传到服务器的/usr/chapter6/webapp 文件中，然后通过 unzip chapter6.war 解压。

6．测试

启动 Tomcat 实例，分别访问如下地址将看到广告内容。

```
http://192.168.1.2:8080/ad?id=1
http://192.168.1.2:8090/ad?id=1
```

7．Nginx 配置

编辑/usr/chapter6/nginx_chapter6.conf 配置文件。

```
upstream backend {
   server 127.0.0.1:8080 max_fails=5 fail_timeout=10s weight=1 backup=
false;
   server 127.0.0.1:8090 max_fails=5 fail_timeout=10s weight=1 backup=
false;
   check interval=3000 rise=1 fall=2 timeout=5000 type=tcp default_down=
false;
   keepalive 100;
}
server {
   listen      80;
   server_name  _;

   location ~ /backend/(.*) {
      keepalive_timeout   30s;
      keepalive_requests  100;

      rewrite /backend(/.*) $1 break;
      #之后该服务将只有内部使用，ngx.location.capture
      proxy_pass_request_headers off;
      #more_clear_input_headers Accept-Encoding;
      proxy_next_upstream error timeout;
      proxy_pass http://backend;
   }
}
```

upstream 配置（http://nginx.org/cn/docs/http/ngx_http_upstream_module.html）有如下项目。

- **server：**指定上游到的服务器。
  - **weight：**权重，权重可以确定负载均衡的比例。
  - **fail_timeout+max_fails：**在指定时间内失败多少次后认为服务器不可用，通过 proxy_next_upstream 来判断是否失败。

- **check**：ngx_http_upstream_check_module模块，上游服务器的健康检查。
  - **interval**：发送心跳包的时间间隔。
  - **rise**：连续成功 rise 次数则认为服务器启动。
  - **fall**：连续失败 fall 次，则认为服务器连接失败。
  - **timeout**：上游服务器请求超时时间。
  - **type**：心跳检测类型（比如此处使用 TCP）。

  更多配置请参考 https://github.com/yaoweibin/nginx_upstream_check_module 和 http://tengine.taobao.org/document_cn/ http_upstream_check_ cn.html。
- **keepalive**：用来支持upstream server http keepalive特性（需要上游服务器支持，比如Tomcat）。默认的负载均衡算法是round-robin，还可以根据IP、URL等通过哈希算法来实现负载均衡。更多资料请参考官方文档。

tomcat keepalive 配置有如下项目（http://tomcat.apache.org/tomcat-7.0-doc/config/http.html）。

- **maxKeepAliveRequests**：默认为100。
- **keepAliveTimeout**：默认等于connectionTimeout，默认为60秒。

location proxy 配置有如下项目（http://nginx.org/cn/docs/http/ngx_http_proxy_module.html）。

- **rewrite**：将当前请求的URL重写，如我们请求时是/backend/ad，则重写后是/ad。
- **proxy_pass**：将整个请求转发到上游服务器。
- **proxy_next_upstream**：负载均衡参数，用来认定什么情况下视为当前upstream server失败，默认是连接失败/超时。
- **proxy_pass_request_headers**：之前已经介绍过，有两个用途，一是如上游服务器不需要请求头，则没必要传输请求头；二是ngx.location.capture时用来防止gzip乱码（也可以使用more_clear_input_headers配置）。
- **keepalive**：keepalive_timeout为keepalive的超时设置，keepalive_requests为长连接数量。此处的keepalive（别人访问该location时的长连接）和upstream keepalive（Nginx与上游服务器的长连接）是不一样的。要注意，如果服务是面向客户的，而且是单个动态内容，则没必要使用长连接。

编辑/usr/servers/nginx/conf/nginx.conf 配置文件。

```
include /usr/chapter6/nginx_chapter6.conf;
#为了方便测试，注释掉 example.conf
```

```
#include /usr/example/example.conf;
```

重启 Nginx。

```
/usr/servers/nginx/sbin/nginx -s reload
```

访问如 192.168.1.2/backend/ad?id=1 即看到结果。可以停掉一个 Tomcat，可以看到服务还是正常的。

在 vim /usr/chapter6/nginx_chapter6.conf 文件中进行配置或修改。

```
location ~ /backend/(.*) {
    internal;
    keepalive_timeout   30s;
    keepalive_requests  1000;
    #支持 keep-alive
    proxy_http_version 1.1;
    proxy_set_header Connection "";

    rewrite /backend(/.*) $1 break;
    proxy_pass_request_headers off;
    #more_clear_input_headers Accept-Encoding;
    proxy_next_upstream error timeout;
    proxy_pass http://backend;
}
```

加上 internal，表示只有内部使用该服务。

20.3.8 Nginx+Lua 逻辑开发

1. 核心代码

编辑/usr/chapter6/ad.lua 代码。

```
local redis = require("resty.redis")
local cjson = require("cjson")
local cjson_encode = cjson.encode
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_re_match = ngx.re.match
local ngx_var = ngx.var

local function close_redis(red)
```

```
    if not red then
        return
    end
    --释放连接（连接池实现）
    local pool_max_idle_time = 10000 --毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)

    if not ok then
        ngx_log(ngx_ERR, "set redis keepalive error : ", err)
    end
end
local function read_redis(id)
    local red = redis:new()
    red:set_timeout(1000)
    local ip = "127.0.0.1"
    local port = 1111
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx_log(ngx_ERR, "connect to redis error : ", err)
        return close_redis(red)
    end

    local resp, err = red:get(id)
    if not resp then
        ngx_log(ngx_ERR, "get redis content error : ", err)
        return close_redis(red)
    end
        --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
    end
    close_redis(red)

    return resp
end

local function read_http(id)
    local resp = ngx.location.capture("/backend/ad", {
        method = ngx.HTTP_GET,
        args = {id = id}
    })

    if not resp then
```

```
        ngx_log(ngx_ERR, "request error :", err)
        return
    end

    if resp.status ~= 200 then
        ngx_log(ngx_ERR, "request error, status :", resp.status)
        return
    end

    return resp.body
end


--获取 id
local id = ngx_var.id

--从 Redis 获取
local content = read_redis(id)

--如果 Redis 没有，则回源到 Tomcat
if not content then
   ngx_log(ngx_ERR, "redis not found content, back to http, id : ", id)
    content = read_http(id)
end

--如果还没有，则返回 404
if not content then
   ngx_log(ngx_ERR, "http not found content, id : ", id)
   return ngx_exit(404)
end

--输出内容
ngx.print("show_ad(")
ngx_print(cjson_encode({content = content}))
ngx.print(")")
```

将可能经常用的变量做成局部变量，如 local ngx_print = ngx.print。使用 jsonp 方式输出，此处可以将请求 URL 限定为/ad/id 方式，这样的好处是可以尽可能早地识别无效请求。可以走 Nginx 缓存/CDN 缓存，缓存的 key 就是 URL，而不带任何参数，防止那些在 URL 加随机参数后穿透缓存。jsonp 使用固定的回调函数 show_ad()，或者限定几个固定的回调来减少缓存的版本。

在 vim /usr/chapter6/nginx_chapter6.conf 文件中进行配置或修改。

```
location ~ ^/ad/(\d+)$ {
    default_type 'text/html';
    charset utf-8;
    lua_code_cache on;
    set $id $1;
    content_by_lua_file /usr/chapter6/ad.lua;
}
```

2. 重启 Nginx

```
/usr/servers/nginx/sbin/nginx -s reload
```

访问如 http://192.168.1.2/ad/1 即可得到结果。而且注意观察日志，第一次访问时不命中 Redis，回源到 Tomcat。第二次请求时就会命中 Redis 了。

第一次访问时，将看到/usr/servers/nginx/logs/error.log 输出类似如下的内容，而第二次请求相同的 URL 不再有如下内容。

```
redis not found content, back to http, id : 2
```

至此整个架构就介绍完了，此处可以直接不使用 Tomcat，而是 Lua 直连 MySQL 做回源处理。另外，本文只是介绍了大体架构，还有更多业务及运维上的细节需要在实际应用中根据自己的场景进行摸索。后续如使用 LVS/HAProxy 做负载均衡、使用 CDN 等可自行搜索相关学习资料进行学习。

本文节选自笔者的开源电子书《跟我学 OpenResty（Nginx+Lua）开发》的第 6 章，相关基础知识可扫二维码进行参考。

21 使用OpenResty开发商品详情页

在第 16 章中已经介绍了设计商品详情页的整体架构和要点，本章将以京东商品详情页为例讲解如何开发商品详情页。京东商品详情页虽然仅是单个页面，但是，其数据聚合源是非常多的，除了一些实时性要求比较高的如价格、库存、服务支持等通过 AJAX 异步加载之外，其他的数据都是在后端做数据聚合，然后拼装网页模板。

如上图所示，商品页主要包括商品基本信息（基本信息、图片列表、颜色/尺码关系、扩展属性、规格参数、包装清单、售后保障等）、商品介绍、其他信息（分类、品牌、店铺（第三方卖家）、店内分类（第三方卖家）、同类相关品牌）。更多细节此处就不再详细阐述。

整个京东有数亿商品，如果每次都要动态获取如上内容进行模板拼装，那么数据来源之多将使性能无法满足要求。最初的解决方案是生成静态页，但是，静态页的最大问题是无法迅速响应页面需求变更，很难做多版本线上对比测试。如上两个因素足以制约商品页的多样化发展，因此，静态化技术不是很好的方案。

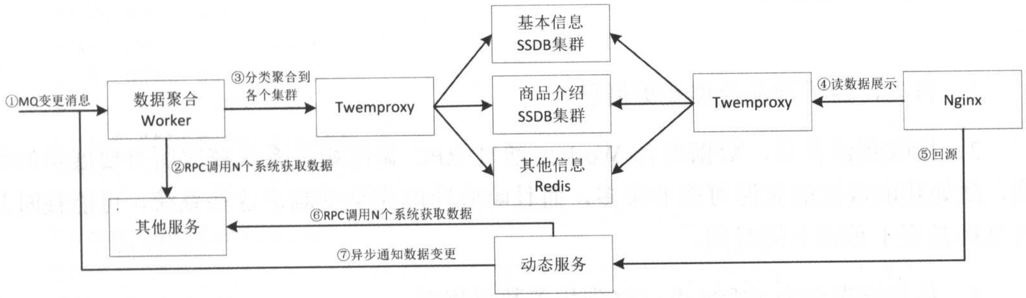
通过分析，数据主要分为四种：商品页基本信息、商品介绍（异步加载）、其他信息（分类、品牌、店铺等）、其他需要实时展示的数据（价格、库存等）。而其他信息如分类、品牌、店铺是非常少的，完全可以放到一个占用内存很小的 Redis 中存储。而商品基本信息可以借鉴静态化技术将数据做聚合存储，数据是原子的，而模板是随时可变的，这样的好处是吸收了静态页聚合的优点，弥补了静态页的多版本缺点。另外一个非常严重的问题就是严重依赖相关系统，如果它们无法正常运行或响应慢，则商品页就会直接受到影响。商品介绍也通过 AJAX 技术惰性加载（因为是第二屏，只有当用户滚动鼠标到该屏时才显示）。而实时展示数据通过 AJAX 技术做异步加载。因此可以做如下设计。

1．接收商品变更消息，做商品基本信息的聚合，即从多个数据源获取商品相关信息，如图片列表、颜色尺码、规格参数、扩展属性等，聚合为一个大的 JSON 数据做成数据闭环，以 key-value 存储。因为是闭环，所以即使依赖的系统无法运行，商品页还能继续服务，对商品页不会造成任何影响。

2. 接收商品介绍变更消息，存储商品介绍信息。

3. 介绍其他信息变更消息，存储其他信息。

整个架构如下图所示。



21.1 技术选型

MQ 可以使用如 Apache ActiveMQ。

Worker/动态服务可以通过如 Java 技术实现。

RPC 可以选择 Alibaba Dubbo。

KV 持久化存储可以选择 SSDB（如果使用 SSD 盘，则可以选择 SSDB+RocksDB 引擎）或者 ARDB（LMDB 引擎版）。

缓存使用 Redis。

SSDB/Redis 分片使用 Twemproxy，这样不管使用 Java 还是 OpenResty（Nginx+Lua），它们都不关心分片逻辑。

前端模板拼装使用 OpenResty。

数据集群数据存储的机器可以采用 RAID 技术或者主从模式防止单点故障。因数据变更不频繁，可以考虑用 SSD 替代 HDD。

21.2 核心流程

1．首先，监听商品数据变更消息。

2．接收到消息后，数据聚合 Worker 通过 RPC 调用相关系统获取所有要展示的数据，此处获取数据的来源可能非常多，而且响应速度完全受制于这些系统，可能耗时几百毫秒甚至 1 秒以上的时间。

3．将数据聚合为 JSON 串存储到相关数据集群。

4．前端 Nginx 通过 Lua 获取相关集群的数据进行展示。商品页需要获取基本信息和其他信息进行模板拼装，即拼装模板仅需要两次调用（另外，因为其他信息数据量少且对一致性要求不高，因此，完全可以缓存到 Nginx 本地全局内存，这样可以减少远程调用、提高性能）。当页面滚动到商品介绍页面时，异步调用商品介绍服务获取数据。

5．如果从聚合的 SSDB 集群/Redis 中获取不到相关数据，则回源到动态服务，通过 RPC 调用相关系统获取所有要展示的数据返回（此处可以做限流处理，因为（如果同一时间请求过多，那么可能导致服务雪崩，所以需要采取保护措施），此处的逻辑和数据聚合 Worker 完全一样。然后发送 MQ 通知数据变更，这样下次访问时就可以从聚合的 SSDB 集群/Redis 中获取数据了。

基本流程如上所述，主要分为 Worker、动态服务、数据存储和前端展示。因为系统非常复杂，只介绍动态服务和前端展示、数据存储架构。Worker 部分不做实现。

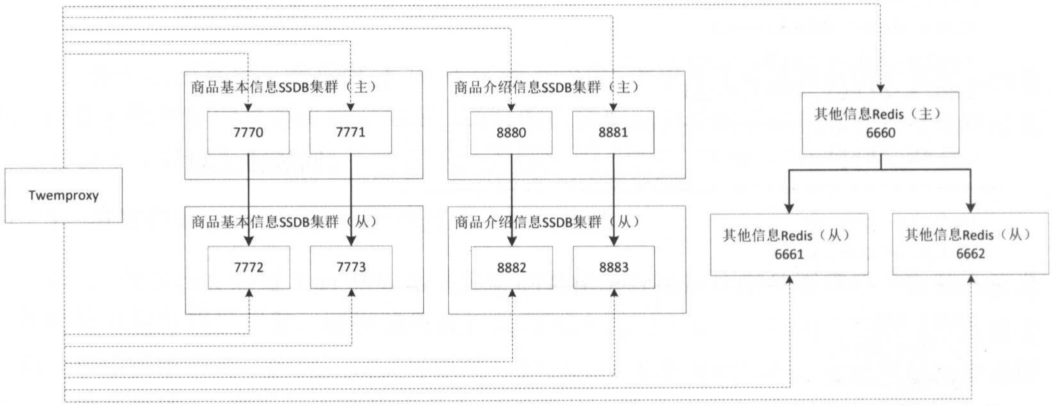
21.3 项目搭建

项目部署目录结构。

```
/usr/chapter7
--------------------------------------------
  ssdb_basic_7770.conf
  ssdb_basic_7771.conf
  ssdb_basic_7772.conf
  ssdb_basic_7773.conf
  ssdb_desc_8880.conf
  ssdb_desc_8881.conf
  ssdb_desc_8882.conf
  ssdb_desc_8883.conf
  redis_other_6660.conf
  redis_other_6661.conf
--------------------------------------------
  nginx_chapter7.conf
--------------------------------------------
  nutcracker.yml
  nutcracker.init
--------------------------------------------
  item.html
  header.html
  footer.html
  item.lua
  desc.lua
  lualib
    item.lua
    item
      common.lua
--------------------------------------------
  webapp
WEB-INF
   lib
   classes
   web.xml
 --------------------------------------------
```

为了演示需要，将各种配置文件都打包到一个目录下，包括 SSDB、Redis、OpenResty、OpenResty 项目、Web 项目。

21.4 数据存储实现



如上图所示，整体架构为主从模式，写数据到主集群，读数据从从集群读取数据，这样当一个集群不足以支撑流量时可以使用更多的集群来支撑更多的访问量。集群分片使用 Twemproxy 实现。

21.4.1 商品基本信息 SSDB 集群配置

编辑/usr/chapter7/ssdb_basic_7770.conf 配置文件。

```
work_dir = /usr/data/ssdb_7770
pidfile = /usr/data/ssdb_7770.pid

server:
        ip: 0.0.0.0
        port: 7770
        allow: 127.0.0.1
        allow: 192.168

replication:
        binlog: yes
        sync_speed: -1
        slaveof:
```

```
logger:
	level: error
	output: /usr/data/ssdb_7770.log
	rotate:
		size: 1000000000

leveldb:
	cache_size: 500
	block_size: 32
	write_buffer_size: 64
	compaction_speed: 1000
	compression: yes
```

编辑/usr/chapter7/ssdb_basic_7771.conf 配置文件。

```
work_dir = /usr/data/ssdb_7771
pidfile = /usr/data/ssdb_7771.pid

server:
	ip: 0.0.0.0
	port: 7771
	allow: 127.0.0.1
	allow: 192.168

replication:
	binlog: yes
	sync_speed: -1
	slaveof:
logger:
	level: error
	output: /usr/data/ssdb_7771.log
	rotate:
		size: 1000000000

leveldb:
	cache_size: 500
	block_size: 32
	write_buffer_size: 64
	compaction_speed: 1000
	compression: yes
```

编辑/usr/chapter7/ssdb_basic_7772.conf 配置文件。

```
work_dir = /usr/data/ssdb_7772
pidfile = /usr/data/ssdb_7772.pid
```

```
server:
      ip: 0.0.0.0
      port: 7772
      allow: 127.0.0.1
      allow: 192.168

replication:
      binlog: yes
      sync_speed: -1
      slaveof:
            type: sync
            ip: 127.0.0.1
            port: 7770

logger:
      level: error
      output: /usr/data/ssdb_7772.log
      rotate:
            size: 1000000000

leveldb:
      cache_size: 500
      block_size: 32
      write_buffer_size: 64
      compaction_speed: 1000
      compression: yes
```

编辑/usr/chapter7/ssdb_basic_7773.conf配置文件。

```
work_dir = /usr/data/ssdb_7773
pidfile = /usr/data/ssdb_7773.pid

server:
      ip: 0.0.0.0
      port: 7773
      allow: 127.0.0.1
      allow: 192.168

replication:
      binlog: yes
      sync_speed: -1
      slaveof:
            type: sync
            ip: 127.0.0.1
```

```
        port: 7771

logger:
    level: error
    output: /usr/data/ssdb_7773.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

配置文件使用 Tab 而不是用空格做缩排，(复制到配置文件后请把空格替换为 Tab)。主从关系：7770（主）→7772（从），7771（主）→7773（从）。配置文件如何配置请参考 https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.md。

创建工作目录。

```
mkdir -p /usr/data/ssdb_7770
mkdir -p /usr/data/ssdb_7771
mkdir -p /usr/data/ssdb_7772
mkdir -p /usr/data/ssdb_7773
```

启动。

```
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ ssdb_basic_7770.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ ssdb_basic_7771.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_basic_7772.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ ssdb_basic_7773.conf &
```

通过 ps -aux | grep ssdb 命令查看是否启动，tail -f nohup.out 查看错误信息。

21.4.2 商品介绍 SSDB 集群配置

编辑/usr/chapter7/ssdb_desc_8880.conf 配置文件。

```
work_dir = /usr/data/ssdb_8880
pidfile = /usr/data/ssdb8880.pid

server:
    ip: 0.0.0.0
    port: 8880
```

```
        allow: 127.0.0.1
        allow: 192.168

replication:
        binlog: yes
        sync_speed: -1
        slaveof:
logger:
        level: error
        output: /usr/data/ssdb_8880.log
        rotate:
                size: 1000000000

leveldb:
        cache_size: 500
        block_size: 32
        write_buffer_size: 64
        compaction_speed: 1000
        compression: yes
```

编辑/usr/chapter7/ssdb_desc_8881.conf 配置文件。

```
work_dir = /usr/data/ssdb_8881
pidfile = /usr/data/ssdb8881.pid

server:
        ip: 0.0.0.0
        port: 8881
        allow: 127.0.0.1
        allow: 192.168

logger:
        level: error
        output: /usr/data/ssdb_8881.log
        rotate:
                size: 1000000000

leveldb:
        cache_size: 500
        block_size: 32
        write_buffer_size: 64
        compaction_speed: 1000
        compression: yes
```

编辑/usr/chapter7/ssdb_desc_8882.conf 配置文件。

```
work_dir = /usr/data/ssdb_8882
pidfile = /usr/data/ssdb_8882.pid

server:
        ip: 0.0.0.0
        port: 8882
        allow: 127.0.0.1
        allow: 192.168

replication:
        binlog: yes
        sync_speed: -1
        slaveof:
replication:
        binlog: yes
        sync_speed: -1
        slaveof:
                type: sync
                ip: 127.0.0.1
                port: 8880

logger:
        level: error
        output: /usr/data/ssdb_8882.log
        rotate:
                size: 1000000000

leveldb:
        cache_size: 500
        block_size: 32
        write_buffer_size: 64
        compaction_speed: 1000
        compression: yes
```

编辑/usr/chapter7/ssdb_desc_8883.conf 配置文件。

```
work_dir = /usr/data/ssdb_8883
pidfile = /usr/data/ssdb_8883.pid

server:
        ip: 0.0.0.0
        port: 8883
        allow: 127.0.0.1
        allow: 192.168
```

```
replication:
	binlog: yes
	sync_speed: -1
	slaveof:
		type: sync
		ip: 127.0.0.1
		port: 8881

logger:
	level: error
	output: /usr/data/ssdb_8883.log
	rotate:
		size: 1000000000

leveldb:
	cache_size: 500
	block_size: 32
	write_buffer_size: 64
	compaction_speed: 1000
	compression: yes
```

配置文件使用 Tab 而不是用空格做缩排（复制到配置文件后请把空格替换为 Tab）。主从关系：7770（主）→7772（从），7771（主）→7773（从）。配置文件如何配置请参考 https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.md。

创建工作目录。

```
mkdir -p /usr/data/ssdb_888{0,1,2,3}
```

启动。

```
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ ssdb_desc_8880.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ ssdb_desc_8881.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8882.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8883.conf &
```

通过 ps -aux | grep ssdb 命令查看是否启动，tail -f nohup.out 查看错误信息。

21.4.3 其他信息 Redis 配置

编辑/usr/chapter7/redis_6660.conf 配置文件。

```
port 6660
pidfile "/var/run/redis_6660.pid"
#设置内存大小，根据实际情况设置，此处测试仅设置 20MB
```

```
maxmemory 20mb
#内存不足时，所有 KEY 按照 LRU 算法删除
maxmemory-policy allkeys-lru
#Redis 的过期算法不是精确的，而是通过采样来算的，默认采样为 3 个，此处我们改成 10
maxmemory-samples 10
#不进行 RDB 持久化
save ""
#不进行 AOF 持久化
appendonly no
```

编辑/usr/chapter7/redis_6661.conf 配置文件。

```
port 6661
pidfile "/var/run/redis_6661.pid"
#设置内存大小，根据实际情况设置，此处测试仅设置 20MB
maxmemory 20mb
#内存不足时，所有 key 按照 LRU 算法进行删除
maxmemory-policy allkeys-lru
#Redis 的过期算法不是精确的，而是通过采样来算的，默认采样为 3 个，此处我们改成 10
maxmemory-samples 10
#不进行 RDB 持久化
save ""
#不进行 AOF 持久化
appendonly no
#主从
slaveof 127.0.0.1 6660
```

编辑/usr/chapter7/redis_6662.conf 配置文件。

```
port 6662
pidfile "/var/run/redis_6662.pid"
#设置内存大小，根据实际情况设置，此处测试仅设置 20MB
maxmemory 20mb
#内存不足时，所有 key 按照 LRU 算法进行删除
maxmemory-policy allkeys-lru
#Redis 的过期算法不是精确的，而是通过采样来算的，默认采样为 3 个，此处我们改成 10
maxmemory-samples 10
#不进行 RDB 持久化
save ""
#不进行 AOF 持久化
appendonly no
#主从
slaveof 127.0.0.1 6660
```

如上配置放到配置文件最末尾即可。此处内存不足时的驱逐算法为所有 key 按照 LRU 进行删除（实际上内存基本上不会遇到满的情况）。主从关系：6660（主）→6661

（从）和 6660（主）→6662（从）。

启动。

```
nohup /usr/servers/redis-2.8.19/redis-server /usr/chapter7/redis_6660.conf &
nohup /usr/servers/redis-2.8.19/redis-server /usr/chapter7/redis_6661.conf &
nohup /usr/servers/redis-2.8.19/redis-server /usr/chapter7/redis_6662.conf &
```

通过 ps -aux | grep redis 命令查看是否启动，tail -f nohup.out 查看错误信息。

21.4.4 集群测试

测试时在主 SSDB/Redis 中写入数据，然后从从 SSDB/Redis 能读取到数据，即表示配置主从成功。

测试商品基本信息 SSDB 集群。

```
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 7770
127.0.0.1:7770> set i 1
OK
127.0.0.1:7770>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 7772
127.0.0.1:7772> get i
"1"
```

测试商品介绍 SSDB 集群。

```
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 8880
127.0.0.1:8880> set i 1
OK
127.0.0.1:8880>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 8882
127.0.0.1:8882> get i
"1"
```

测试其他信息集群。

```
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 6660
127.0.0.1:6660> set i 1
OK
```

```
127.0.0.1:6660> get i
"1"
127.0.0.1:6660>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli  -p 6661
127.0.0.1:6661> get i
"1"
```

21.4.5 Twemproxy 配置

编辑/usr/chapter7/nutcracker.yml 配置文件。

```
basic_master:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:7770:1 server1
   - 127.0.0.1:7771:1 server2

basic_slave:
  listen: 127.0.0.1:1112
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:7772:1 server1
   - 127.0.0.1:7773:1 server2

desc_master:
  listen: 127.0.0.1:1113
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:8880:1 server1
```

```
    - 127.0.0.1:8881:1 server2

desc_slave:
  listen: 127.0.0.1:1114
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  servers:
   - 127.0.0.1:8882:1 server1
   - 127.0.0.1:8883:1 server2

other_master:
  listen: 127.0.0.1:1115
  hash: fnv1a_64
  distribution: random
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:6660:1 server1


other_slave:
  listen: 127.0.0.1:1116
  hash: fnv1a_64
  distribution: random
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:6661:1 server1
   - 127.0.0.1:6662:1 server2
```

- 因为使用了主从，所以需要给Server起一个名字，如server1、server2。否则，分片算法默认根据ip:port:weight，这样就会使主从数据的分片算法不一致。
- 因为其他信息Redis是单实例全量，没有进行分片，因此分片算法可以使用random。
- 我们使用了hash_tag，可以保证相同的tag在一个分片上（本例配置了，但没有用到该特性）。

复制《跟我学 OpenResty（Nginx+Lua）开发》第 6 章的 nutcracker.init，并把配置文件改为 usr/chapter7/nutcracker.yml。然后通过/usr/chapter7/nutcracker.init start 启动 Twemproxy。

测试主从集群是否工作正常，代码如下。

```
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1111
127.0.0.1:1111> set i 1
OK
127.0.0.1:1111>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1112
127.0.0.1:1112> get i
"1"
127.0.0.1:1112>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1113
127.0.0.1:1113> set i 1
OK
127.0.0.1:1113>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1114
127.0.0.1:1114> get i
"1"
127.0.0.1:1114>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1115
127.0.0.1:1115> set i 1
OK
127.0.0.1:1115>
root@kaitao:/usr/chapter7#
/usr/servers/redis-2.8.19/src/redis-cli -p 1116
127.0.0.1:1116> get i
"1"
```

到此数据集群配置成功。

21.5 动态服务实现

因为真实数据是从多个子系统获取，很难模拟这么多子系统交互，所以此处使用假数据来进行实现。

21.5.1 项目搭建

使用 Maven 搭建 Web 项目，Maven 知识请自行学习。

21.5.2 项目依赖

本文将最小化依赖，即仅依赖需要的 Servlet、Jackson、Guava、Jedis。

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>17.0</version>
  </dependency>
  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.5.2</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.3.3</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.3.3</version>
  </dependency>
</dependencies>
```

guava 是类似于 apache commons 的一个基础类库，用于简化一些重复操作，可以参考 http://ifeve.com/google-guava/。

21.5.3 核心代码

编辑 com.github.zhangkaitao.chapter7.servlet.ProductServiceServlet 代码。

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String type = req.getParameter("type");
        String content = null;
        try {
            if("basic".equals(type)) {
                content = getBasicInfo(req.getParameter("skuId"));
            } else if("desc".equals(type)) {
                content = getDescInfo(req.getParameter("skuId"));
            } else if("other".equals(type)) {
                content = getOtherInfo(req.getParameter("ps3Id"),
                        req. getParameter("brandId"));
            }
        } catch (Exception e) {
            e.printStackTrace();

resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            return;
        }
        if(content != null) {
            resp.setCharacterEncoding("UTF-8");
            resp.getWriter().write(content);
        } else {
            resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
        }
    }
```

根据请求参数 type 来决定调用哪个服务获取数据。

1. 基本信息服务

```
    private String getBasicInfo(String skuId) throws Exception {
        Map<String, Object> map = new HashMap<String, Object>();
        //商品编号
        map.put("skuId", skuId);
        //名称
        map.put("name", "苹果（Apple）iPhone 6 (A1586) 16GB 金色 移动联通电
                信 4G 手机");
        //一级二级三级分类
        map.put("ps1Id", 9987);
        map.put("ps2Id", 653);
        map.put("ps3Id", 655);
        //品牌 ID
        map.put("brandId", 14026);
```

```
        //图片列表
        map.put("imgs", getImgs(skuId));
        //上架时间
        map.put("date", "2014-10-09 22:29:09");
        //商品毛重
        map.put("weight", "400");
        //颜色尺码
        map.put("colorSize", getColorSize(skuId));
        //扩展属性
        map.put("expands", getExpands(skuId));
        //规格参数
        map.put("propCodes", getPropCodes(skuId));
        map.put("date", System.currentTimeMillis());
        String content = objectMapper.writeValueAsString(map);
        //实际应用应该是发送 MQ
        asyncSetToRedis(basicInfoJedisPool, "p:" + skuId + ":", content);
        return objectMapper.writeValueAsString(map);
    }

    private List<String> getImgs(String skuId) {
        return Lists.newArrayList(
            "jfs/t277/193/1005339798/768456/29136988/542d0798N19d42ce3.jpg",
            "jfs/t352/148/1022071312/209475/53b8cd7f/542d079bN3ea45c98.jpg",
            "jfs/t274/315/1008507116/108039/f70cb380/542d0799Na03319e6.jpg",
            "jfs/t337/181/1064215916/27801/b5026705/542d079aNf184ce18.jpg"
        );
    }

    private List<Map<String, Object>> getColorSize(String skuId) {
        return Lists.newArrayList(
            makeColorSize(1217499, "金色", "公开版（16GB ROM）"),
            makeColorSize(1217500, "深空灰", "公开版（16GB ROM）"),
            makeColorSize(1217501, "银色", "公开版（16GB ROM）"),
            makeColorSize(1217508, "金色", "公开版（64GB ROM）"),
            makeColorSize(1217509, "深空灰", "公开版（64GB ROM）"),
            makeColorSize(1217509, "银色", "公开版（64GB ROM）"),
            makeColorSize(1217493, "金色", "移动 4G 版 （16GB）"),
            makeColorSize(1217494, "深空灰", "移动 4G 版 （16GB）"),
            makeColorSize(1217495, "银色", "移动 4G 版 （16GB）"),
            makeColorSize(1217503, "金色", "移动 4G 版 （64GB）"),
            makeColorSize(1217503, "金色", "移动 4G 版 （64GB）"),
            makeColorSize(1217504, "深空灰", "移动 4G 版 （64GB）"),
            makeColorSize(1217505, "银色", "移动 4G 版 （64GB）")
        );
```

```
    }
    private Map<String, Object> makeColorSize(long skuId, String color,
String size) {
        Map<String, Object> cs1 = Maps.newHashMap();
        cs1.put("SkuId", skuId);
        cs1.put("Color", color);
        cs1.put("Size", size);
        return cs1;
    }

    private List<List<?>> getExpands(String skuId) {
        return Lists.newArrayList(
                (List<?>)Lists.newArrayList("热点", Lists.newArrayList("超
                薄 7mm 以下", "支持 NFC")),
                (List<?>)Lists.newArrayList("系统", "苹果（IOS）"),
                (List<?>)Lists.newArrayList("系统", "苹果（IOS）"),
                (List<?>)Lists.newArrayList("购买方式", "非合约机")
        );
    }

    private Map<String, List<List<String>>> getPropCodes(String skuId) {
        Map<String, List<List<String>>> map = Maps.newHashMap();
        map.put("主体", Lists.<List<String>>newArrayList(
                Lists.<String>newArrayList("品牌", "苹果（Apple）"),
                Lists.<String>newArrayList("型号", "iPhone 6 A1586"),
                Lists.<String>newArrayList("颜色", "金色"),
                Lists.<String>newArrayList("上市年份", "2014 年")
        ));
        map.put("存储", Lists.<List<String>>newArrayList(
                Lists.<String>newArrayList("机身内存", "16GB ROM"),
                Lists.<String>newArrayList("储存卡类型", "不支持")
        ));
        map.put("显示", Lists.<List<String>>newArrayList(
                Lists.<String>newArrayList("屏幕尺寸", "4.7 英寸"),
                Lists.<String>newArrayList("触摸屏", "Retina HD"),
                Lists.<String>newArrayList("分辨率", "1334 x 750")
        ));
        return map;
    }
```

本例基本信息提供了如商品名称、图片列表、颜色尺码、扩展属性、规格参数等数据。而为了简化逻辑，大多数数据都是 List/Map 数据结构。

2. 商品介绍服务

```
    private String getDescInfo(String skuId) throws Exception {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("content",  "<div><img  data-lazyload='http://  img30.
360buyimg.com/jgsq-productsoa/jfs/t448/127/574781110/103911/b3c80634/5472
ba22N45400f4e.jpg' alt='' /><img data-lazyload='http:// img30.360buyimg.
com/jgsq-productsoa/jfs/t802/133/19465528/162152/e463e43/54e2b34aN11bceb7
0.jpg' alt='' height='386' width='750' /></div>");
        map.put("date", System.currentTimeMillis());
        String content = objectMapper.writeValueAsString(map);
        //实际应用应该是发送 MQ
        asyncSetToRedis(descInfoJedisPool, "d:" + skuId + ":", content);
        return objectMapper.writeValueAsString(map);
    }
```

3. 其他信息服务

```
    private String getOtherInfo(String ps3Id, String brandId) throws
Exception {
        Map<String, Object> map = new HashMap<String, Object>();
        //面包屑
        List<List<?>> breadcrumb = Lists.newArrayList();
        breadcrumb.add(Lists.newArrayList(9987, "手机"));
        breadcrumb.add(Lists.newArrayList(653, "手机通讯"));
        breadcrumb.add(Lists.newArrayList(655, "手机"));
        //品牌
        Map<String, Object> brand = Maps.newHashMap();
        brand.put("name", "苹果（Apple）");
        brand.put("logo",
"BrandLogo/g14/M09/09/10/rBEhVlK6vdkIAAAAAAFLXzp-lIAAHWawP_QjwAAAVF472.png");
        map.put("breadcrumb", breadcrumb);
        map.put("brand", brand);
        //实际应用应该是发送 MQ
        asyncSetToRedis(otherInfoJedisPool, "s:" + ps3Id + ":",
                    objectMapper.writeValueAsString(breadcrumb));
        asyncSetToRedis(otherInfoJedisPool, "b:" + brandId + ":",
                    objectMapper.writeValueAsString(brand));
        return objectMapper.writeValueAsString(map);
    }
```

本例中其他信息只使用了面包屑和品牌数据。

4. 辅助工具

```
private ObjectMapper objectMapper = new ObjectMapper();
private JedisPool basicInfoJedisPool = createJedisPool("127.0.0.1", 1111);
private JedisPool descInfoJedisPool = createJedisPool("127.0.0.1", 1113);
private JedisPool otherInfoJedisPool = createJedisPool("127.0.0.1", 1115);

private JedisPool createJedisPool(String host, int port) {
    GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();
    poolConfig.setMaxTotal(100);
    return new JedisPool(poolConfig, host, port);
}

private ExecutorService executorService = Executors.newFixedThreadPool(10);
private void asyncSetToRedis(final JedisPool jedisPool, final String
                                key, final String content) {
    executorService.submit(new Runnable() {
        @Override
        public void run() {
            Jedis jedis = null;
            try {
                jedis = jedisPool.getResource();
                jedis.set(key, content);
            } catch (Exception e) {
                e.printStackTrace();
                jedisPool.returnBrokenResource(jedis);
            } finally {
                jedisPool.returnResource(jedis);
            }

        }
    });
}
```

本例使用 Jackson 进行 JSON 的序列化。Jedis 进行 Redis 的操作。使用线程池做异步更新（实际应用中，可以使用 MQ 做实现）。

21.5.4 web.xml 配置

```
<servlet>
    <servlet-name>productServiceServlet</servlet-name>

<servlet-class>com.github.zhangkaitao.chapter7.servlet.ProductServiceServ
let</servlet-class>
```

```
</servlet>
<servlet-mapping>
    <servlet-name>productServiceServlet</servlet-name>
    <url-pattern>/info</url-pattern>
</servlet-mapping>
```

21.5.5 打 WAR 包

```
cd D:\workspace\chapter7
mvn clean package
```

此处使用 maven 命令打包，比如本例将得到 chapter7.war，然后将其上传到服务器的/usr/chapter7/webapp，然后通过 unzip chapter6.war 解压。

21.5.6 配置 Tomcat

复制《跟我学 OpenResty（Nginx+Lua）开发》第 6 章使用的 Tomcat 实例。

```
cd /usr/servers/
cp -r tomcat-server1 tomcat-chapter7/
vim /usr/servers/tomcat-chapter7/conf/Catalina/localhost/ROOT.xml
```

```
<!-- 访问路径是根，Web 应用所属目录为/usr/chapter7/webapp -->
<Context path="" docBase="/usr/chapter7/webapp"></Context>
```

指向第 7 章的 Web 应用路径。

21.5.7 测试

启动 Tomcat 实例。

```
/usr/servers/tomcat-chapter7/bin/startup.sh
```

访问如下 URL 进行测试。

```
http://192.168.1.2:8080/info?type=basic&skuId=1
http://192.168.1.2:8080/info?type=desc&skuId=1
http://192.168.1.2:8080/info?type=other&ps3Id=1&brandId=1
```

21.5.8 Nginx 配置

编辑/usr/chapter7/nginx_chapter7.conf 配置文件。

```
upstream backend {
```

```
        server 127.0.0.1:8080 max_fails=5 fail_timeout=10s weight=1;
        check interval=3000 rise=1 fall=2 timeout=5000 type=tcp default_down
=false;
        keepalive 100;
    }

    server {
        listen       80;
        server_name  item2015.jd.com item.jd.com d.3.cn;

        location ~ /backend/(.*) {
            #internal;
            keepalive_timeout   30s;
            keepalive_requests  1000;
            #支持 keep-alive
            proxy_http_version 1.1;
            proxy_set_header Connection "";

            rewrite /backend(/.*) $1 break;
            proxy_pass_request_headers off;
            #more_clear_input_headers Accept-Encoding;
            proxy_next_upstream error timeout;
            proxy_pass http://backend;
        }
    }
```

此处 server_name 指定了 item.jd.com（商品详情页）和 d.3.cn（商品介绍）。其他配置可以参考第 6 章的内容。另外，实际生产环境要把#internal 打开，表示只有本 Nginx 能访问。

编辑/usr/servers/nginx/conf/nginx.conf 配置文件。

```
include /usr/chapter7/nginx_chapter7.conf;
#为了方便测试，注释掉 example.conf
include /usr/chapter6/nginx_chapter6.conf;
#lua 模块路径，其中";;"表示默认搜索路径，默认到/usr/servers/nginx 下找
lua_package_path "/usr/chapter7/lualib/?.lua;;";  #lua 模块
lua_package_cpath "/usr/chapter7/lualib/?.so;;";  #c 模块
```

Lua 模块从/usr/chapter7 目录加载，因为我们要使用自己写的模块。

重启 Nginx。

/usr/servers/nginx/sbin/nginx -s reload

21.5.9 绑定 hosts 测试

192.168.1.2 item.jd.com

192.168.1.2 item2015.jd.com

192.168.1.2 d.3.cn

访问 http://item.jd.com/backend/info?type=basic&skuId=1 即可看到结果。

21.6 前端展示实现

分为三部分实现：基础组件、商品介绍、前端展示部分。

21.6.1 基础组件

首先，进行基础组件的实现，商品介绍和前端展示部分都需要读取 Redis 和 HTTP 服务，因此，可以抽取公共部分出来复用。

编辑/usr/chapter7/lualib/item/common.lua 代码。

```
local redis = require("resty.redis")
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local function close_redis(red)
    if not red then
        return
    end
    --释放连接(连接池实现)
    local pool_max_idle_time = 10000 --毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)

    if not ok then
        ngx_log(ngx_ERR, "set redis keepalive error : ", err)
    end
end
```

```
local function read_redis(ip, port, keys)
    local red = redis:new()
    red:set_timeout(1000)
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx_log(ngx_ERR, "connect to redis error : ", err)
        return close_redis(red)
    end
    local resp = nil
    if #keys == 1 then
        resp, err = red:get(keys[1])
    else
        resp, err = red:mget(keys)
    end
    if not resp then
        ngx_log(ngx_ERR, "get redis content error : ", err)
        return close_redis(red)
    end

    --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
    end
    close_redis(red)

    return resp
end

local function read_http(args)
    local resp = ngx.location.capture("/backend/info", {
        method = ngx.HTTP_GET,
        args = args
    })

    if not resp then
        ngx_log(ngx_ERR, "request error")
        return
    end
    if resp.status ~= 200 then
        ngx_log(ngx_ERR, "request error, status :", resp.status)
        return
    end
    return resp.body
end
```

```
local _M = {
    read_redis = read_redis,
    read_http = read_http
}
return _M
```

整个逻辑和第 6 章类似。只是 read_redis 根据参数 keys 个数支持 get 和 mget。比如，read_redis(ip, port, {"key1"})，则调用 get，而 read_redis(ip, port, {"key1", "key2"})，则调用 mget。

21.6.2 商品介绍

1. 核心代码

编辑/usr/chapter7/desc.lua 代码。

```
local common = require("item.common")
local read_redis = common.read_redis
local read_http = common.read_http
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_re_match = ngx.re.match
local ngx_var = ngx.var

local descKey = "d:" .. skuId .. ":"
local descInfoStr = read_redis("127.0.0.1", 1114, {descKey})
if not descInfoStr then
   ngx_log(ngx_ERR, "redis not found desc info, back to http, skuId : ",
skuId)
   descInfoStr = read_http({type="desc", skuId = skuId})
end
if not descInfoStr then
   ngx_log(ngx_ERR, "http not found basic info, skuId : ", skuId)
   return ngx_exit(404)
end
ngx_print("showdesc(")
ngx_print(descInfoStr)
ngx_print(")")
```

通过复用逻辑后，整体代码简化了许多。此处从集群读取商品介绍。另外，前端展

示使用 JSONP 技术展示商品介绍。

2. Nginx 配置

编辑/usr/chapter7/nginx_chapter7.conf 配置文件。

```
location ~^/desc/(\d+)$ {
    if ($host != "d.3.cn") {
       return 403;
    }
    default_type application/x-javascript;
    charset utf-8;
    lua_code_cache on;
    set $skuId $1;
    content_by_lua_file /usr/chapter7/desc.lua;
}
```

因为 item.jd.com 和 d.3.cn 复用了同一个配置文件，此处需要限定只有 d.3.cn 域名能访问，以防止恶意访问。

重启 Nginx 后，访问 http://d.3.cn/desc/1 即可得到 JSONP 结果。

21.6.3 前端展示

1. 核心代码

编辑/usr/chapter7/item.lua 代码。

```
local common = require("item.common")
local item = require("item")
local read_redis = common.read_redis
local read_http = common.read_http
local cjson = require("cjson")
local cjson_decode = cjson.decode
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_var = ngx.var


local skuId = ngx_var.skuId

--获取基本信息
```

```
    local basicInfoKey = "p:" .. skuId .. ":"
    local basicInfoStr = read_redis("127.0.0.1", 1112, {basicInfoKey})
    if not basicInfoStr then
      ngx_log(ngx_ERR, "redis not found basic info, back to http, skuId : ",
skuId)
      basicInfoStr = read_http({type="basic", skuId = skuId})
    end
    if not basicInfoStr then
      ngx_log(ngx_ERR, "http not found basic info, skuId : ", skuId)
      return ngx_exit(404)
    end

    local basicInfo = cjson_decode(basicInfoStr)
    local ps3Id = basicInfo["ps3Id"]
    local brandId = basicInfo["brandId"]
    --获取其他信息
    local breadcrumbKey = "s:" .. ps3Id .. ":"
    local brandKey = "b:" .. brandId ..":"
    local otherInfo = read_redis("127.0.0.1", 1116, {breadcrumbKey, brandKey})
or {}
    local breadcrumbStr = otherInfo[1]
    local brandStr = otherInfo[2]
    if breadcrumbStr then
      basicInfo["breadcrumb"] = cjson_decode(breadcrumbStr)
    end
    if brandStr then
      basicInfo["brand"] = cjson_decode(brandStr)
    end
    if not breadcrumbStr and not brandStr then
      ngx_log(ngx_ERR, "redis not found other info, back to http, skuId : ",
brandId)
      local otherInfoStr = read_http({type="other", ps3Id = ps3Id, brandId
= brandId})
      if not otherInfoStr then
         ngx_log(ngx_ERR, "http not found other info, skuId : ", skuId)
      else
        local otherInfo = cjson_decode(otherInfoStr)
        basicInfo["breadcrumb"] = otherInfo["breadcrumb"]
        basicInfo["brand"] = otherInfo["brand"]
      end
    end

    local name = basicInfo["name"]
    --name to unicode
```

```
basicInfo["unicodeName"] = item.utf8_to_unicode(name)
--字符串截取，超长显示...
basicInfo["moreName"] = item.trunc(name, 10)
--初始化各分类的 URL
item.init_breadcrumb(basicInfo)
--初始化扩展属性
item.init_expand(basicInfo)
--初始化颜色尺码
item.init_color_size(basicInfo)
local template = require "resty.template"
template.caching(true)
template.render("item.html", basicInfo)
```

整个逻辑分为四部分：获取基本信息，根据基本信息中的关联关系获取其他信息，初始化/格式化数据，渲染模板。

2. 初始化模块

编辑/usr/chapter7/lualib/item.lua 代码。

```
local bit = require("bit")
local utf8 = require("utf8")
local cjson = require("cjson")
local cjson_encode = cjson.encode
local bit_band = bit.band
local bit_bor = bit.bor
local bit_lshift = bit.lshift
local string_format = string.format
local string_byte = string.byte
local table_concat = table.concat

--utf8 转为 unicode
local function utf8_to_unicode(str)
    if not str or str == "" or str == ngx.null then
        return nil
    end
    local res, seq, val = {}, 0, nil
    for i = 1, #str do
        local c = string_byte(str, i)
        if seq == 0 then
            if val then
                res[#res + 1] = string_format("%04x", val)
            end

           seq = c < 0x80 and 1 or c < 0xE0 and 2 or c < 0xF0 and 3 or
```

```
            c < 0xF8 and 4 or --c < 0xFC and 5 or c < 0xFE and 6 or 0
             if seq == 0 then
                 ngx.log(ngx.ERR, 'invalid UTF-8 character sequence'
.. ",,," .. tostring(str))
                 return str
             end

             val = bit_band(c, 2 ^ (8 - seq) - 1)
         else
             val = bit_bor(bit_lshift(val, 6), bit_band(c, 0x3F))
         end
         seq = seq - 1
     end
     if val then
         res[#res + 1] = string_format("%04x", val)
     end
     if #res == 0 then
         return str
     end
     return "\\u" .. table_concat(res, "\\u")
 end

 --utf8 字符串截取
 local function trunc(str, len)
    if not str then
      return nil
    end

    if utf8.len(str) > len then
      return utf8.sub(str, 1, len) .. "..."
    end
    return str
 end

 --初始化面包屑
 local function init_breadcrumb(info)
     local breadcrumb = info["breadcrumb"]
     if not breadcrumb then
        return
     end

     local ps1Id = breadcrumb[1][1]
     local ps2Id = breadcrumb[2][1]
     local ps3Id = breadcrumb[3][1]
```

```
        --此处应该根据一级分类查找 url
        local ps1Url = "http://shouji.jd.com/"
        local ps2Url = "http://channel.jd.com/shouji.html"
        local ps3Url = "http://list.jd.com/list.html?cat=" .. ps1Id .. ",
" .. ps2Id .. "," .. ps3Id

        breadcrumb[1][3] = ps1Url
        breadcrumb[2][3] = ps2Url
        breadcrumb[3][3] = ps3Url
    end

    --初始化扩展属性
    local function init_expand(info)
      local expands = info["expands"]
      if not expands then
        return
      end
      for _, e in ipairs(expands) do
        if type(e[2]) == "table" then
          e[2] = table_concat(e[2], ", ")
        end
      end
    end

    --初始化颜色尺码
    local function init_color_size(info)
      local colorSize = info["colorSize"]

      --颜色尺码 JSON 串
      local colorSizeJson = cjson_encode(colorSize)
      --颜色列表（不重复）
      local colorList = {}
      --尺码列表（不重复）
      local sizeList = {}
      info["colorSizeJson"] = colorSizeJson
      info["colorList"] = colorList
      info["sizeList"] = sizeList

      local colorSet = {}
      local sizeSet = {}
      for _, cz in ipairs(colorSize) do
        local color = cz["Color"]
        local size = cz["Size"]
```

```
        if color and color ~= "" and not colorSet[color] then
          colorList[#colorList + 1] = {color = color,
url = "http://item.jd.com/" ..cz["SkuId"] .. ".html"}
          colorSet[color] = true
        end
        if size and size ~= "" and not sizeSet[size] then
          sizeList[#sizeList + 1] = {size = size,
url = "http://item.jd.com/" ..cz["SkuId"] .. ".html"}
          sizeSet[size] = ""
        end
      end
    end

    local _M = {
      utf8_to_unicode = utf8_to_unicode,
      trunc = trunc,
      init_breadcrumb = init_breadcrumb,
      init_expand = init_expand,
      init_color_size = init_color_size
    }

    return _M
```

utf8_to_unicode 代码之前已经见过了，其他的都是一些逻辑代码。

3. 模板 HTML 片段

编辑/usr/chapter7/item.html 文件。

```
        var pageConfig = {
            compatible: true,
            product: {
                skuid: {* skuId *},
                name: '{* unicodeName *}',
                skuidkey:'AFC266E971535B664FC926D34E91C879',
                href: 'http://item.jd.com/{* skuId *}.html',
                src: '{* imgs[1] *}',
                cat: [{* ps1Id *},{* ps2Id *},{* ps3Id *}],
                brand: {* brandId *},
                tips: false,
                pType: 1,
                venderId:0,
                shopId:'0',
specialAttrs:["HYKHSP-0","isDistribution","isHaveYB","isSelfService-0","i
```

```
sWeChatStock-0","packType","IsNewGoods","isCanUseDQ","isSupportCard","isC
anUseJQ","isOverseaPurchase-0","is7ToReturn-1","isCanVAT"],
                videoPath:'',
                desc: 'http://d.3.cn/desc/{* skuId *}'
             }
          };
          var warestatus = 1;
          {% if colorSizeJson then %} var ColorSize = {* colorSizeJson *};{%
end %}
```

{* var *}输出变量，{% code %} 写代码片段。

面包屑

```
<div class="breadcrumb">
    <strong>
        <a href='{* breadcrumb[1][3] *}'>{* breadcrumb[1][2] *}</a></ strong>
    <span>
         &gt; 
        <a href='{* breadcrumb[2][3] *}'>{* breadcrumb[2][2] *}</a>
         &gt; 
        <a href='{* breadcrumb[3][3] *}'>{* breadcrumb[3][2] *}</a>
         &gt; 
    </span>
    <span>
        {% if brand then %}
        <a href='http://www.jd.com/pinpai/{* ps3Id *}-{* brandId *}.
html'>{* brand['name'] *}</a>
         &gt; 
       {% end %}
       <a href='http://item.jd.com/{* skuId *}.html'>{* moreName *}</a>
    </span>
</div>
```

图片列表

```
<div id="spec-n1" class="jqzoom" onclick="window.open('http:// www.jd.
com/bigimage.aspx?id={* skuId *}')" clstag="shangpin|keycount |product|
spec-n1">
    <img data-img="1" width="350" height="350" src="http://img14.
360buyimg.com/n1/{* imgs[1] *}" alt="{* name *}"/>
</div>
<div id="spec-list" clstag="shangpin|keycount|product|spec-n5">
    <a href="javascript:;" class="spec-control" id="spec-forward"></a>
    <a href="javascript:;" class="spec-control" id="spec-backward"></a>
    <div class="spec-items">
```

```
            <ul class="lh">
                {% for _, img in ipairs(imgs) do %}
                <li><img class='img-hover' alt='{* name *}' src='http://
img14.360buyimg.com/n5/{* img *}' data-url='{* img *}' data-img='1'
width='50' height='50'></li>
                {% end %}
            </ul>
        </div>
    </div>
```

颜色尺码选择

```
    <div class="dt">选择颜色：</div>
        <div class="dd">
            {% for _, color in ipairs(colorList) do %}
                <div class="item"><b></b><a href="{* color['url'] *}" title=
"{* color['color'] *}"><i>{* color['color'] *}</i></a></div>
            {% end %}
        </div>
    </div>
    <div id="choose-version" class="li">
        <div class="dt">选择版本：</div>
        <div class="dd">
            {% for _, size in ipairs(sizeList) do %}
                <div class="item"><b></b><a href="{* size['url'] *}" title=
"{* size['size'] *}">{* size['size'] *}</a></div>
            {% end %}
        </div>
    </div>
```

扩展属性

```
    <ul id="parameter2" class="p-parameter-list">
        <li title='{* name *}'>商品名称：{* name *}</li>
        <li title='{* skuId *}'>商品编号：{* skuId *}</li>
        {% if brand then %}
        <li title='{* brand["name"] *}'>品牌：<a href='http://www.jd.com/pinpai/{*
ps3Id *}-{* brandId *}.html' target='_blank'>{* brand["name"] *}</a></li>
        {% end %}
        {% if date then %}
        <li title='{* date *}'>上架时间：{* date *}</li>
        {% end %}
        {% if weight then %}
        <li title='{* weight *}'>商品毛重：{* weight *}</li>
        {% end %}
```

```
    {% for _, e in pairs(expands) do %}
    <li title='{* e[2] *}'>{* e[1] *}: {* e[2] *}</li>
    {% end %}
</ul>
```

规格参数

```
<table cellpadding="0" cellspacing="1" width="100%" border="0" class="Ptable">
    {% for group, pc in pairs(propCodes) do %}
    <tr><th class="tdTitle" colspan="2">{* group *}</th><tr>
    {% for _, v in pairs(pc) do %}
    <tr><td class="tdTitle">{* v[1] *}</td><td>{* v[2] *}</td></tr>
    {% end %}
    {% end %}
</table>
```

4. Nginx 配置

编辑/usr/chapter7/nginx_chapter7.conf 配置文件。

```
#模板加载位置
set $template_root "/usr/chapter7";
location ~ ^/(\d+).html$ {
    if ($host !~ "^(item|item2015)\.jd\.com$") {
      return 403;
    }
    default_type 'text/html';
    charset utf-8;
    lua_code_cache on;
    set $skuId $1;
    content_by_lua_file /usr/chapter7/item.lua;
}
```

21.6.4 测试

重启 Nginx，访问 http://item.jd.com/1217499.html 可得到响应内容，本例和京东的商品详情页的数据有些出入，因为输出的页面做了一些精简。

21.6.5 优化

1. local cache

对于数据一致性要求不敏感，而且数据量很少的其他信息，完全可以在本地缓存全量。而且可以设置 5~10 分钟的过期时间，这是完全可以接受的。因此，可以使用

lua_shared_dict 全局内存进行缓存。具体逻辑可以参考如下代码。

```
local nginx_shared = ngx.shared
--item.jd.com 配置的缓存
local local_cache = nginx_shared.item_local_cache
local function cache_get(key)
   if not local_cache then
      return nil
   end
   return local_cache:get(key)
end

local function cache_set(key, value)
   if not local_cache then
      return nil
   end
   return local_cache:set(key, value, 10 * 60) --10 分钟
end

local function get(ip, port, keys)
   local tables = {}
   local fetchKeys = {}
   local resp = nil
   local status = STATUS_OK
   --如果 tables 是个 map，则 #tables 拿不到长度
   local has_value = false
   --先读取本地缓存
   for i, key in ipairs(keys) do
      local value = cache_get(key)
      if value then
         if value == "" then
            value = nil
         end
         tables[key] = value
         has_value = true
      else
         fetchKeys[#fetchKeys + 1] = key
      end
   end

   --如果还有数据没获取，则从 Redis 获取
   if #fetchKeys > 0 then
      if #fetchKeys == 1 then
         status, resp = redis_get(ip, port, fetchKeys[1])
```

```
        else
            status, resp = redis_mget(ip, port, fetchKeys)
        end
        if status == STATUS_OK then
            for i = 1, #fetchKeys do
                local key = fetchKeys[i]
                local value = nil
                if #fetchKeys == 1 then
                    value = resp
                else
                    value = get_data(resp, i)
                end
                tables[key] = value
                 has_value = true
                 cache_set(key, value or "", ttl)
            end
        end
    end
    --如果从缓存查到，那么就认为ok
    if has_value and status == STATUS_NOT_FOUND then
        status = STATUS_OK
    end
    return status, tables
end
```

2. nginx proxy cache

为了防止恶意刷页面或热点页面访问频繁，可以使用 nginx proxy_cache 做页面缓存。当然，还可以选择 Apache Traffic Server、Squid、Varnish 等做内容缓存。

nginx.conf 缓存配置

```
    proxy_buffering on;
    proxy_buffer_size 8k;
    proxy_buffers 256 8k;
    proxy_busy_buffers_size 64k;
    proxy_temp_file_write_size 64k;
    proxy_temp_path /usr/servers/nginx/proxy_temp;
    #设置Web缓存区名称为cache_one，内存缓存空间大小为200MB，1分钟没有被访问的
    #内容自动清除，硬盘缓存空间大小为30GB。
    proxy_cache_path  /usr/servers/nginx/proxy_cache levels=1:2
keys_zone=cache_item:200m inactive=1m max_size=30g;
```

增加 proxy_cache 的配置，可以通过挂载一块内存作为缓存的存储空间。更多配置

规则请参考 http://nginx.org/cn/docs/http/ngx_http_proxy_module.html。

nginx_chapter7.conf 配置

与 server 指令配置同级。

```
############ 测试时使用的动态请求
map $host $item_dynamic {
   default                  "0";
   item2015.jd.com           "1";
}
```

即如果域名为 item2015.jd.com，则 item_dynamic=1。

```
   location ~ ^/(\d+).html$ {
      set $skuId $1;
      if ($host !~ "^(item|item2015)\.jd\.com$") {
        return 403;
      }

      expires 3m;
      proxy_cache cache_item;
      proxy_cache_key $uri;
      proxy_cache_bypass $item_dynamic;
      proxy_no_cache $item_dynamic;
      proxy_cache_valid 200 301 3m;
      proxy_cache_use_stale updating error timeout invalid_header
http_500 http_502 http_503 http_504;
      proxy_pass_request_headers off;
      proxy_set_header Host $host;
      #支持 keep-alive
      proxy_http_version 1.1;
      proxy_set_header Connection "";
      proxy_pass http://127.0.0.1/proxy/$skuId.html;
      add_header X-Cache '$upstream_cache_status';
   }

   location ~ ^/proxy/(\d+).html$ {
      allow 127.0.0.1;
      deny all;
      keepalive_timeout  30s;
      keepalive_requests  1000;
      default_type 'text/html';
      charset utf-8;
      lua_code_cache on;
```

```
    set $skuId $1;
    content_by_lua_file /usr/chapter7/item.lua;
}
```

expires：设置响应缓存头信息，此处是 3min。将会得到 Cache-Control:max-age=180 和类似 Expires:Sat, 28 Feb 2015 10:01:10 GMT 的响应头。

proxy_cache：使用之前在 nginx.conf 中配置的 cache_item 缓存。

proxy_cache_key：缓存 key 为 URI，不包括 Host 和参数，这样不管用户怎么通过在 URL 上加随机数都是可以缓存的。

proxy_cache_bypass：Nginx 不从缓存读取响应的条件，可以写多个。如果存在一个字符串条件且不是“0”，那么 Nginx 就不会从缓存中读取响应内容。此处，如果使用的 host 为 item2015.jd.com，那么就不会从缓存读取响应内容。

proxy_no_cache：Nginx 不将响应内容写入缓存的条件，可以写多个。如果存在一个字符串条件且不是“0”，那么 Nginx 就不会将响应内容写入缓存。此处，如果使用的 host 为 item2015.jd.com，那么就不会将响应内容写入缓存。

proxy_cache_valid：为不同的响应状态码设置不同的缓存时间，此处对 200、301 缓存 3min。

proxy_cache_use_stale：什么情况下使用不新鲜（过期）的缓存内容。配置和 proxy_next_upstream 内容类似。此处配置了如果出现连接出错、超时、404、500 等问题，都会使用不新鲜的缓存内容。此外我们配置了 updating 配置，通过配置它可以在 Nginx 更新缓存（其中一个 Worker 进程）时（其他的 Worker 进程）使用不新鲜的缓存进行响应，这样可以减少回源的数量。

proxy_pass_request_headers：不需要请求头，所以不传递。

proxy_http_version 1.1 和 proxy_set_header Connection ""：支持 keepalive。

add_header X-Cache '$upstream_cache_status'：添加是否缓存命中的响应头。比如命中 HIT、不命中 MISS、不走缓存 BYPASS。或者命中会看到 X-Cache：HIT 响应头。

allow/deny：允许和拒绝访问的 IP 列表，此处只允许本机访问。

keepalive_timeout 30s 和 keepalive_requests 1000：支持 keepalive。

nginx_chapter7.conf 清理缓存配置。

```
    location /purge {
        allow    127.0.0.1;
        allow    192.168.0.0/16;
        deny     all;
        proxy_cache_purge  cache_item $arg_url;
    }
```

只允许内网访问。访问如 http://item.jd.com/purge?url=/11.html。如果看到 Successful purge，则说明缓存存在并已经清理了。

修改 item.lua 代码

```
--添加 Last-Modified，用于响应 304 缓存
ngx.header["Last-Modified"] = ngx.http_time(ngx.now())

local template = require "resty.template"
template.caching(true)
template.render("item.html", basicInfo)
```

在渲染模板前设置 Last-Modified，用于判断内容是否变更，默认 Nginx 通过等于去比较，也可以通过配置 if_modified_since 指令来支持小于等于比较。如果请求头发送的 If-Modified-Since 和 Last-Modified 匹配，则返回 304 响应，即内容没有变更，使用本地缓存。此处可能看到了 Last-Modified 是当前时间，不是商品信息变更时间。商品信息变更时间由商品信息变更时间、面包屑变更时间和品牌变更时间三者决定，因此，实际应用时应该取三者中最晚的时间。还有一个问题就是模板内容可能变了，但是，商品信息没有变，此时使用 Last-Modified 得到的内容可能是错误的，所以可以通过使用 ETag 技术来解决这个问题，ETag 可以认为是内容的一个摘要，内容变更后摘要就变了。

3. GZIP 压缩

修改 nginx.conf 配置文件。

```
    gzip on;
    gzip_min_length  4k;
    gzip_buffers    4 16k;
    gzip_http_version 1.0;
    gzip_proxied any;
    #前端是 squid 的情况下要加此参数，否则 squid 上不缓存 gzip 文件
    gzip_comp_level 2;
    gzip_types       text/plain application/x-javascript text/css
application/xml;
    gzip_vary on;
```

此处指定数据至少 4k 时才进行压缩，如果数据太小，则压缩没有意义。

至此整个商品详情页逻辑就介绍完了，一些细节和运维内容需要在实际开发中实际处理，无法做到面面俱到。

本章内容节选自笔者的开源电子书《跟我学 OpenResty（Nginx+Lua）开发》第 7 章，相关基础知识可扫二维码进行参考。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

经历618、双11多次大考，保证大规模电商系统高流量、高频次的葵花宝典。

徐春俊，京东集团副总裁/京东保险业务负责人

集中火力讲述作者构建京东大流量系统用到的高可用和高并发原则。

马松，京东集团副总裁/京东商城研发体系负责人

浓缩作者多年对网站系统升级迭代的创新、技术、实践和积累。

肖军，京东集团副总裁/京东X事业部负责人

有高可用和高并发总体原则、关键技术、实战经验的总结，更有曾经踩过的坑。

杨建，京东保险高级研发总监

教你如何构建高并发、大流量系统方能经受起亿级线上用户流量的真实考验。

王晓钟，京东商城高级研发总监

从前端到DB底层设计，本书无不精细阐述。

尚鑫，京东商城研发总监

站在一个新高度思考网站后台技术，从应用级缓存到前端缓存，从SOA到闭环。

杨思勇，京东商城研发总监

京东多年架构升级及大促备战的高质量总结。

王彪，京东商城研发总监

将系统设计的深奥套路讲得如此清晰，难能可贵。

付彩宝，京东商城研发总监

完整呈现如何设计响应亿级请求的京东商品详情页系统。

王春明，京东商城研发总监

将技术应用于业务、理论应用于实践的名著。

何小锋 京东商城基础平台首席架构师

地表至强，天大福利。

者文明，京东商城运营研发部首席架构师

流量并发暴增与系统架构变革的十字路口，需要它。

鲍永成，京东商城容器引擎平台负责人

一个亿级流量网站和一个中小型网站的技术架构难度截然不同。

陈锋，京东云平台事业部架构师

这种指导手册式的技术书籍，值得精读和细品。

赵云霄 京东商城 API网关 负责人

一本互联网高并发架构设计的百科全书。

李尊敬，京东商城交易平台架构师

从各角度剖析系统设计的优化要点和注意事项。

赵辉，京东商城交易平台架构师

循序渐进地将一系列复杂问题阐述得清晰、易读。

尤凤凯，京东商城交易平台架构师

实战出真理，选择这本书，靠谱。

刘峻桦，京东商城网站平台架构师

博文视点Broadview

新浪微博
weibo.com
@博文视点Broadview

上架建议：网站架构

策划编辑：张春雨
责任编辑：徐津平
封面设计：李　玲

ISBN 978-7-121-30954-0
9 787121 309540 >
定价：99.00元